UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Mirjana Domazet-Lošo

ALGORITHMS FOR EFFICIENT ALIGNMENT-FREE SEQUENCE COMPARISON
ALGORITMI ZA UČINKOVITU USPOREDBU SEKVENCI BEZ KORIŠTENJA SRAVNJIVANJA

DOCTORAL THESIS
DOKTORSKA DISERTACIJA

Zagreb, 2010.

The research related to this doctoral dissertation was conducted at the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, and within the Bioinformatics group, Department of Evolutionary Genetics, Max Planck Institute for Evolutionary Biology, Plön, Germany.

Supervisors: Prof. Dr. Bernhard Haubold, Max Planck Institute for Evolutionary Biology, Plön, Germany

Asst. Prof. Dr. Strahil Ristov, Ruđer Bošković Institute, Zagreb, Croatia

Number of Pages: 127

Dissertation Number:

Table of Contents

1. GENERAL INTRODUCTION 1

1.1. Suffix trees and other index data structures used in biological sequence analysis.. 9

1.1.1. Suffix Tree 11

1.1.2. The space and the time complexity of the algorithms for the suffix tree construction 13

1.1.3. Suffix Array 14

1.1.4. The space and the time complexity of the algorithms for suffix array construction 15

1.1.5. Enhanced Suffix Array 17

1.1.6. The 64-bit implementation of the lightweight suffix array construction algorithm21

1.1.7. Self-index 22

1.1.8. Burrows-Wheeler transform 23

1.1.9. The FM-Index and the backward search algorithm 25

1.1.10. The space and the time-complexity of the FM-index 29

2. EFFICIENT ESTIMATION OF PAIRWISE DISTANCES BETWEEN GENOMES 31

2.1. Introduction.. 31

2.2. Methods 33

2.2.1. Definition of an alignment-free estimator of the rate of substitution, K_r 33

2.2.2. Problem statement 35

2.2.3. Time complexity analysis of the previous approach (kr 1)................ 35

2.2.4. Time complexity analysis of the new approach (kr 2) 37

2.2.5. Algorithm 1: Computation of all K_r values during the traversal of a generalized suffix tree of n sequences.. 38

2.2.6. The implementation of kr version 2 44

2.3. Analysis of K_r on simulated data sets 45

2.3.1. Auxiliary programs 45

2.3.2. Consistency of K_r 46

2.3.3. The affect of horizontal gene transfer on the accuracy of K_r *48*
2.3.4. The effect of genome duplication on the accuracy of K_r.................... *49*
2.3.5. Run time comparison of kr 1 and kr 2 .... *50*
2.4. Application of *kr* version 2 53
2.4.1. Auxililary software used for the analysis of real data sets *56*
2.4.2. The analysis of 12 Drosophila genomes *57*
2.4.3. The analysis of 13 Escherichia coli and Shigella genomes *58*
2.4.4. The analysis of 825 HIV-1 pure subtype genomes............................. *61*
2.5. Discussion 62

3. EFFICIENT ALIGNMENT-FREE DETECTION OF LOCAL SEQUENCE HOMOLOGY.. 66
3.1. Introduction.. 66
3.2. Methods 69
3.2.1. Problem statement – determining subtype(s) of a query sequence.... *69*
3.2.2. Construction of locally homologous segments *71*
3.2.3. Time complexity of computing a list of intervals I_i *72*
3.2.4. Algorithm 2: Construction of an interval tree *73*
3.2.5. Computing a list of segements G_i... *80*
3.3. Analysis of *st* on simulated data sets 82
3.3.1. Run-time and memory usage analysis of st....................................... *82*
3.3.2. Consistency of st *85*
3.3.3. Comparison to SCUEAL on simulated data sets *92*
3.4. Application of *st*.. 97
3.4.1. The analysis of Neisseria meningitidis *98*
3.4.2. The analysis of a recombinant form of HIV-1 .... *99*
3.4.3. The analysis of circulating recombinant forms of HIV-1 .... *103*
3.4.4. The analysis of an avian pathogenic Escherichia coli strain *104*
3.5. Discussion 107

4. CONCLUSION 110
5. REFERENCES 112
6. ELECTRONIC SOURCES.. 121
7. LIST OF ABBREVIATIONS AND SYMBOLS 122

ABSTRACT **124**
SAŽETAK **125**
CURRICULUM VITAE **126**
ŽIVOTOPIS **127**

1. General Introduction

Nucleic acids (DNA and RNA) are the molecular carriers of hereditary information in living organisms. These molecules are polymers of four nucleotides. DNA sequences are usually represented as strings of characters over the alphabet {A, C, G, T}, where each character corresponds to a nucleotide base. Similarly, RNA sequences are represented as strings over the alphabet {A, C, G, U}. The size of the DNA or RNA of an organism (its genome size) varies significantly. For example, the size of viral genomes ranges from few thousand to around million nucleotide base pairs; bacterial genomes sizes range from hundreds of thousands to less than ten million base pairs, and the size of mammalian genomes ranges between one and eight billion base pairs (Gregory, 2005), e.g. a human genome comprises around 3 billion base pairs.

All the existing and the extinct genomes are the outcome of the copying process that happens each generation from the emergence of the first living cell approximately 3.8 billion years ago. However, this process was accompanied by mutations and recombination. The genetic variation thus generated permitted adaptation to different habitats, which resulted in the diversity of present and extinct organisms. Thus, the evolution of organisms or sequences can be envisaged as a branching process where every pair of organisms or sequences has a common ancestor at a varying depth of an emerging tree.

However, evolution is a historical process, which usually cannot be observed directly. Therefore, we reconstruct this branching process from the physically available sequences. The first step in this endeavor is to compare different sequences and to find similar regions between them. In particular, the number of unique combinations is extremely large even for very short sequences; for example, a DNA sequence comprising only 100 base pairs (which is less than 0.00001% of a human genome) can have $4^{100} \approx 2 \cdot 10^{60}$ unique combinations. In addition, the space of nucleotide sequences occupied by present day and extinct organisms represents only a small portion of all possible combinations. Hence, it is possible to construct hypotheses about common ancestry of sequences (also called homology) using

sequence similarity scores which are above the score expected by chance alone (Karlin and Altschul, 1990).

Historical branching patterns (also known as phylogenies) are not the only information that can be reconstructed from the comparison of sequences. Regions of high sequence similarity between even relatively distantly-related organisms usually imply similar biological functions or structures. In this way, the functional information, inferred through experiments in one organism, can be transferred to a genome of another organism solely based on sequence comparison between these organisms. The advantage of this approach becomes apparent when we consider that functional experiments in some organisms (e.g. humans, animals with long generation time) are much more difficult if not impossible, than in the so called "model organisms", e.g. yeast, bacteria, fruit flies, mice.

This is why sequence comparison is an essential tool in modern biology: the study of functional and structural organization, evolutionary mechanisms and evolutionary history of organisms all rely upon sequence comparison. But, how can we compare sequences in the first place? The traditional, widely-used approach to sequence comparison is sequence alignment. The goal of this procedure is to find the most plausible hypothesis about homology between nucleotide positions in two or more sequences. For this purpose, sequences are typically arranged in a matrix (Figure 1-1), where each sequence corresponds to a matrix row, and the columns of a matrix represent the homologous nucleotides. Due to mutations, the homologous nucleotides are not always identical. In order to align homologous nucleotides within the same column, gaps (spaces) can be inserted in a sequence. In Figure 1-1, two sequences are aligned in five ways, and for each alignment two scores are computed. The goodness of an alignment is determined based on the alignment score. The optimal alignment score corresponds to the minimal weighted edit distance between sequences, that is, the minimal sum of weighted (scored) edit operations required to transform one sequence into another. Hence, each alignment score is computed as the sum of rewards for matches (identical nucleotides within the same column), and penalties for mismatches (different nucleotides within the same column) and gaps (when a nucleotide is aligned with a gap). In the biological context, a mismatch is observed as a nucleotide substitution (or a point mutation), and a gap as an insertion

or a deletion of a nucleotide (Figure 1-1). An example of a scoring scheme with arbitrary values for matches, mismatches and gaps is shown in Figure 1-1. In this example, there are two scoring systems: similarity score depicts a similarity between strings, in which case the best alignment is the one with the maximal score. The distance score, on the other hand, measures the distance between strings, and here the minimal score corresponds to the optimal alignment. In the example in Figure 1-1, the best results for both scoring schemes are achieved for the first alignment.

	Alignment	Similarity score	Distance score
1	S_1: A C C G S_2: A C G G	2	1
2	S_1: A C C G - S_2: A - C G G	-1	4
3	S_1: A C C G - S_2: - A C G G	-3	5
4	S_1: A C C G - - S_2: - - A C G G	-10	10

Figure 1-1. An Example of Pairwise Sequence Alignment. Sequences S_1 = AC**C**G and S_2 = AC**G**G can be aligned in several possible ways, five of which are listed here. An indel (insertion or deletion) is denoted by a gap (-). The similarity score between a sequence pair is computed based on the following values: each match is rewarded 1, each substitution -1, and a gap -2. For example, for the first alignment, the score is $3 \cdot 1 - 1 = 2$, since there are 3 matches, and a mismatch. The distance score between a sequence pair is computed based on the following values: each match is 0, the penalty for a substitution is 1, and the penalty for a gap is 2. For example, for the first alignment, the score is $3 \cdot 0 + 1 = 1$. Thus, the best alignment score in both cases is obtained for the first alignment.

The alignment procedure can be pair-wise or between multiple sequences. In the first case, a pair of sequences is aligned, and in the second case, more than two sequences are aligned. Furthermore, an alignment procedure can be also global or local. Global sequence alignment is usually applied to similar sequences which are homologous along their entire lengths. Local sequence alignment is usually applied to more divergent sequences, where only some regions are homologous. The optimal alignment, with respect to the chosen scoring scheme, can be found for both global and local alignment. The Needleman-Wunsch algorithm (Needleman and Wunsch, 1970) is a dynamic programming algorithm that constructs the optimal pair-wise global alignment (the chosen similarity scoring scheme can be similar to the example in Figure 1-1). The algorithm requires both $O(|S_1| \cdot |S_2|)$ time and space for the comparison of a sequence pair (S_1, S_2). This time complexity corresponds to the computation of $|S_1| \cdot |S_2|$ matrix entries, where each entry represents a score for a different partial alignment between the first and the second sequence. However, the memory usage of this algorithm can be reduced to $O(|S_1| + |S_2|)$ using Hirschberg's algorithm (Hirschberg, 1975), with a further increase in the run-time. To illustrate the requirements of the optimal global alignment computed by the Needleman-Wunsch algorithm, let us look at an example: a human genome comprises 3 billion nucleotide base pairs, and since each DNA molecule has two strands of the same length, it totals 6 billion nucleotides. If we wish to compare two human genomes using a computer that can execute a billion instructions per second, then the comparison of two human genomes using the Needleman-Wunsch algorithm would require $6 \cdot 10^9 \cdot 6 \cdot 10^9 / 10^9$ = 36 billion seconds ≈ 1142 years to compute all entries in a matrix. Similarly, a variant of the Needleman-Wunsch algorithm, the Smith-Waterman algorithm (Smith and Waterman, 1981), results in the optimal local sequence alignment. Again, the time and the space complexity of the algorithm is $O(|S_1| \cdot |S_2|)$ for a sequence pair (S_1, S_2). Moreover, finding the optimal multiple sequence alignment is an NP-complete problem under a commonly used scoring scheme (Wang and Jiang, 1994). Therefore, many alignment tools are based on some heuristic in order to improve the run-time (e.g. Edgar and Batzoglou, 2006; Section 2.1).

Once an alignment is computed, it still does not directly reflect evolutionary distance. In particular, the evolutionary distance between nucleotide sequences is the

number of nucleotide substitutions per site. Thus, the similarity score between a pair of sequences derived from an alignment should be transformed into evolutionary distance. The simplest model used for this purpose, the Jukes-Cantor model, is based on the idea that each nucleotide base has an equal chance of mutating to any other base (Jukes and Cantor, 1969). The Jukes-Cantor formula (1-1) converts p, the proportion of mismatches between a sequence pair (S_1, S_2), into the number of nucleotide substitutions per site:

$$K(S_1, S_2) = -\frac{3}{4} ln\left(1 - \frac{4}{3} p\right) \qquad (1\text{-}1)$$

Parameter p can be easily obtained from the alignment. In the example in Figure 1-1, the value of p of the first alignment is 0.25 (one mismatch per 4 nucleotides) which yields an evolutionary distance of 0.304 nucleotide substitutions per site. In the computation of the relative number of pair-wise mismatches between sequences, the regions with gaps are usually ignored. In addition, a number of more complex models have been proposed (e.g. Kimura 1980; Felsenstein, 1981). These models are based on more than one parameter for different types of nucleotide conversions. Among them, probably the most commonly used is Kimura's 2-parameter model (Kimura, 1980), which distinguishes between transitions and transversions.

Evolutionary distances can be further used to construct phylogenies (Figure 1-2). A phylogenetic (or evolutionary) tree is usually a bifurcating tree whose leaves represent sequences or organisms. Each internal node (a bifurcation in the tree) corresponds to a common ancestor of two or more entities (organisms or sequences) at the leaves of the tree. There are several methods for the reconstruction of a phylogenetic tree based on evolutionary distances between all sequence pairs: e.g. Neighbor-Joining method (Saitou and Nei, 1987), and UPGMA (Michener and Sokal, 1957). These two methods are based on clustering: first, the two sequences which have the smallest evolutionary distance are joined (that is, their most recent common ancestor is added to the tree), and then the third sequence or a set of sequences, which are closest to them, are added to the tree, and so on. The requirement of this procedure is that the total distance between sequences in the set has to stay minimal. The diagram representing a phylogenetic tree is called

dendrogram. A special case of a dendrogram, where the branch lengths correspond to the number of nucleotide substitutions, is called a phylogram (Figure 1-2c).

a) MSA of S_1, S_2, and S_3

S_1	A	C	C	G
S_2	A	C	G	G
S_3	A	C	T	T

b) Matrix of evolutionary distances between S_1, S_2, and S_3

	S_1	S_2	S_3
S_1	0	0.304	0.824
S_2	0.304	0	0.824
S_3	0.824	0.824	0

c) Phylogenetic tree of S_1, S_2, and S_3

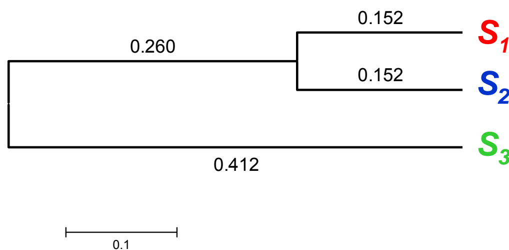


Figure 1-2. Construction of the phylogenetic tree of sequences S_1, S_2, S_3. a) In the first step, the multiple sequence alignment (MSA) of S_1, S_2, and S_3 is constructed. **b)** Next, the evolutionary distances between sequences are determined using the Jukes-Cantor formula (formula (1-1)). These values are computed from the number of pair-wise mismatches between sequence pair, which is 0.25 for (S_1, S_2), and 0.5 for (S_1, S_3) and (S_2, S_3). The evolutionary distance is a symmetric measure, and the values on the diagonal are 0. **c)** The phylogenetic tree is constructed using the Neighbor-Joining method (see text) and drawn in MEGA 4 (Kumar *et al.*, 2008). The evolutionary distance between a sequence pair (S_i, S_j) is the sum of values along branches from S_i to S_j. For example, the distance between S_1 and S_2 is 0.304, which can be computed as the sum of 0.152 plus 0.152.

In order to show the scale of the data that biologists wish to analyze, let us consider the human 1000 genome project (www.1000genomes.org)[1], which started

[1] Another large-scale sequencing project is currently under way: the 1000 plants project (www.onekp.com). In addition, the volume of available nucleotide sequences is constantly increasing:

in 2008, and is expected to finish in 2011. This project involves sequencing of at least 1000 human genomes in order to catalog human genetic variation, which would assist research in genetic diseases. Such a large-scale project was made feasible by the latest sequencing techniques which significantly lowered the sequencing cost (for example, the current estimate of the sequencing costs of this project is 30-50 million dollars, while the application of older sequencing techniques would have required 500 million dollars; see en.wikipedia.org/wiki/1000_Genomes_Project). It is expected that the project will yield at least 6 trillion nucleotides, which will be available to the public. However, the time and the memory requirements of the alignment procedures do not scale well for the comparison of large genomes. As an efficient alternative, alignment-free methods, which were first proposed more than two decades ago (Blaisdell, 1986), might be used for the comparison of very long sequences. It was also noticed that alignment-free methods can perform better on rearranged sequences than global alignment methods (Vinga and Almeida, 2003; Höhl *et al*., 2006). However, in the comparison of syntenic sequences, alignment-based methods outperformed alignment-free methods (Höhl and Ragan, 2007).

Alignment-free methods have been developed in two directions: methods that rely on the analysis of word frequencies between sequences, and methods based on information theory (Vinga and Almeida, 2003). However, the distance measures obtained by alignment-free methods are generally not related to evolutionary models. In an attempt to provide an efficient alignment-free method which produces biologically relevant evolutionary distances, we have recently developed an alignment-free pair-wise distance measure, K_r (Haubold *et al*. 2009). K_r is an estimator of the number of nucleotide substitutions per site, based on the Jukes-Cantor model of DNA sequence evolution (Jukes and Cantor, 1969). As a result, K_r-based phylogenies of closely related genomes are more accurate than phylogenies based on model-free distance measures. However, the first implementation of the method was slow for large samples of genomes.

---

the size of GenBank, a sequence database which contains all publicly available nucleotide sequences, now doubles every 30 months (Benson *et al*., 2009). In 2008, GenBank contained approximately 92 million sequences with over 95 billion base pairs (Benson *et al*., 2009).

In the first part of my thesis (Chapter 2), I address the problem of efficient estimation of K_r pair-wise distances. I developed a new algorithm for the rapid computation of K_r distances (Algorithm 1), which I implemented in the program, *kr* version 2 (Domazet-Lošo and Haubold, 2009). To illustrate the run-time improvement of the new implementation and its applicability, the program was tested on simulated data sets, and a wide range of complete genomes data sets: 825 genomes of HIV-1 strains (7.5 million base pairs), 13 genomes of enterobacteria (over 60 million base pairs), and the complete genomes of 12 *Drosophila* (over 2 billion base pairs) (Domazet-Lošo and Haubold, 2009).

However, in the K_r-based clustering of 825 genomes of HIV-1, a single strain was not classified according to its official subtype. Further analysis of this strain revealed a phylogenetic incongruence along its genome: the strain was a recombinant of at least two different subtypes (see Sections 2.4.4 and 3.4.3). Motivated by this observation, in the second part of my thesis, I investigated the detection of regions of local sequence similarity (Chapter 3). I proposed a solution based on a new algorithm for efficient detection of locally matching regions between sequences (Algorithm 2), and implemented it in the program *st*. To illustrate the efficiency and the scalability of the program, it was used for the classification of strains of the circulating recombinant forms of the human immunodeficiency virus (HIV) (over 3 million base pairs), and for the detection of locally homologous regions between an avian pathogenic *Escherichia coli* strain and a set of 13 enterobacterial strains (over 60 million base pairs).

Both algorithms (Algorithm 1 and Algorithm 2), which I developed as a part of this thesis, rely on the concept of a generalized suffix tree (Gusfield, 1997). Therefore, I begin by giving an overview of suffix trees and similar index data structures, which are widely-used in the analysis of biological sequences (Section 1.1). I address the first problem, efficient estimation of pairwise distances between genomes, in Chapter 2. In Chapter 3, I concentrate on the alignment-free detection of local sequence homology. The thesis conclusions are presented in Chapter 4.

1.1. Suffix trees and other index data structures used in biological sequence analysis

The comparison of two or more sequence generally relies on the detection of conserved islands (exact or similar subsequences) between sequences, or sometimes within a sequence. In particular, the large-scale sequence comparison requires the upmost efficiency in both the time and the memory requirements for the retrieval of these regions. In technical terms, the problem of finding matching regions between sequences corresponds to the classical problem of the pattern search, i.e. finding the occurrences of a (typically short) pattern in a (long) text. This problem can be described by introducing the following notation: Let P and T be strings over the same alphabet, Σ, where the lengths of P and T are denoted as $|P|$ and $|T|$, respectively. P is usually the shorter string, usually referred to as *pattern* (or *query*), and T is the longer string, called *text* (or *subject*), and usually $|T| >> |P|$. The problem of finding exact matches is the problem of finding all z occurrences ($z \geq 0$) of P in T. This problem can be naively solved in the worst case $O(|P| \cdot |T|)$ time. Of course, this is impractical for large text. Hence, the more sophisticated solutions were developed to address this problem. These solutions apply two approaches to speed up the computation: (i) a pattern is indexed, which requires only a small extra space; or (ii) a text is indexed, which requires the significant amount of additional space. The two classical algorithms that index pattern are the Knuth-Morris-Pratt algorithm (Knuth *et al.*, 1977) and the Boyer-Moore algorithm (Boyer and Moore, 1977). They can solve the problem of finding exact matches of P in T in the worst case $O(|T|)$ time, with a preprocessing step which takes $O(|P|)$ time, although the Boyer-Moore algorithm on average performs even better, with $O(|T| / |P|)$ time. However, when several patterns need to be found in the same text, then the $O(|T|)$ time does not scale well.

The other approach is based on the preprocessing (indexing) of the text; in particular, it uses the suffix tree data structure or its variants. The suffix tree is an index data structure, and one of the most important data structures in string algorithms and the analysis of biological sequences (Gusfield, 1997). The suffix tree resembles the book index principle: first, the lexicographically sorted index of a book is built, and then, the location of an arbitrary entry is found by just searching the

index, and not the entire book, with the number of comparisons which is at most equal to the entry length.

The suffix tree data structure can be efficiently used in problems like finding the exact matches, but its real strength is shown in more complicated problems, like finding the longest common substrings between sequences. In the exact matching problem, the suffix tree is constructed for the text T in $O(|T|)$ time, and then all z occurrences of the pattern P are searched in $O(|P| + z)$ time (Section 1.1.2). In the problem of finding the longest common substrings between a sequence pair (S_1, S_2), the suffix tree is constructed in $O(|S_1| + |S_2|)$ time, and then the traversal of the tree in order to find the longest common substring is performed in the same linear time. The following section (1.1.1) describes this widely-used data structure.

The more space efficient counter-part of the suffix tree is the suffix array (Manber and Myers, 1993; Section 1.1.3). However, the memory requirements of both suffix trees and suffix arrays are proportional to the text size. In order to reduce the space required for the index storage, researchers developed self-indexes. A self-index is the data structure that replaces text and requires space proportional to the compressed text (e.g. Navarro and Mäkinen, 2007; Ferragina *et al*., 2008; Section 1.1.7). In contrast, typical string operations performed using self-indexes are slower than in the case of suffix tress and suffix arrays; e.g. searching a pattern in a text is one order of magnitude slower than the same operation performed using a suffix array (Ferragina *et al*., 2008; Välimäki *et al*., 2007). Moreover, current implementations of self-indexes require significant additional space (linear in the text size) for the construction of indexes (Ferragina *et al*., 2008).

In a quest for the best trade-off between the time and the space requirements needed for the index construction in the programs which I developed as a part of my thesis, I used the enhanced suffix array (Kasai *et al*., 2001; Abouelhoda *et al*., 2004; Section 1.1.5), which is an extension of the suffix array, as the underlying data structure to implement the suffix tree concept (Section 1.1.6).

1.1.1. Suffix Tree

S*uffix tree* is a data structure developed for efficient operations on strings (Gusfield, 1997). It is a rooted directed tree that represents all suffixes of a string (Figure 1-3e).

Let T be the suffix tree for a string S. Let S be defined over the alphabet Σ. Every suffix s_i of S is defined as $s_i = S[i..|S|]$, and $1 \leq i \leq |S|$. A tree *leaf* (or a *terminal node*), denoted (S, i), corresponds to a suffix s_i of S. Every *edge* (or *branch*) of T is labeled with characters. Every branch label represents a substring of S. The concatenation of characters from the root of T to a leaf (S, i) of T spells out a suffix s_i of S. Every *internal node* (or *branch node*) x of T, except a root, has at least two, and maximally $|\Sigma|$ branches. The label on each branch of x starts with a different character from Σ. A path-label from the root of T to an internal node x of T represents a common substring of two or more suffixes of S. The *string depth* of x is the length of the path-label of x.

A *generalized suffix tree* is a data structure that represents every suffix of a set of strings.

Example 1.1

Let S = ACCA\$. The suffixes of S are s_1 = ACCA\$, s_2 = CCA\$, s_3 = CA\$, s_4 = A\$, and s_5 = \$. The character \$ is a sentinel character that is not part of the alphabet over which S is built, so it does not exist anywhere else in S. The terminal character ensures that every suffix ends as a leaf. If the character \$ were not added to the end of S, then the suffix s_4 would be s_4 = A, and would not end up as a leaf of T, but as an internal node. However, the convention of adding \$ at the end of the string is usually not required in the implementation.

Figure 1-3 illustrates the naïve construction of T built for string S, which leads to $O(|S|^2)$ construction. In the first step (Figure 1-3a), the leaf $(S, 1)$, corresponding to the suffix s_1, is connected to the root of T (node x_1). In the second step (Figure 1-3b), the suffix s_2 is added to the tree. At most $|s_2|$ comparisons are needed to add the suffix s_2 to the non-empty tree. The first character of s_2, C, is compared to the starting characters on the labels of branches starting at the root of T. Since the label

of the only existing branch starts with A, the comparison finishes here. A new branch is added for s_2, which connects the leaf (S, 2) to the root of the tree (x_1).

In the third step, s_3 is added to the tree (Figure 1-3c). The first character of s_3 is compared to the characters on the labels of already existing branches. Since a branch starting with character C already exists, the new suffix follows this branch. The second character of s_3 is compared to the second character on the branch label. Since they differ, the branch labeled with CCA$ in Figure 1-3b is split in two, and a new branch node (x_2) is added at that position. The new branches connect leaves (S, 2) and (S, 3), corresponding to the suffixes s_2 and s_3 respectively, to the branch node x_2.

Similarly, the fourth suffix, s_4, is added to the tree (here, the new branch node is x_3) (Figure 1-3d). Finally, the suffix s_5 is added (Figure 1-3e). For this suffix, starting with the terminal character, a new branch is added.

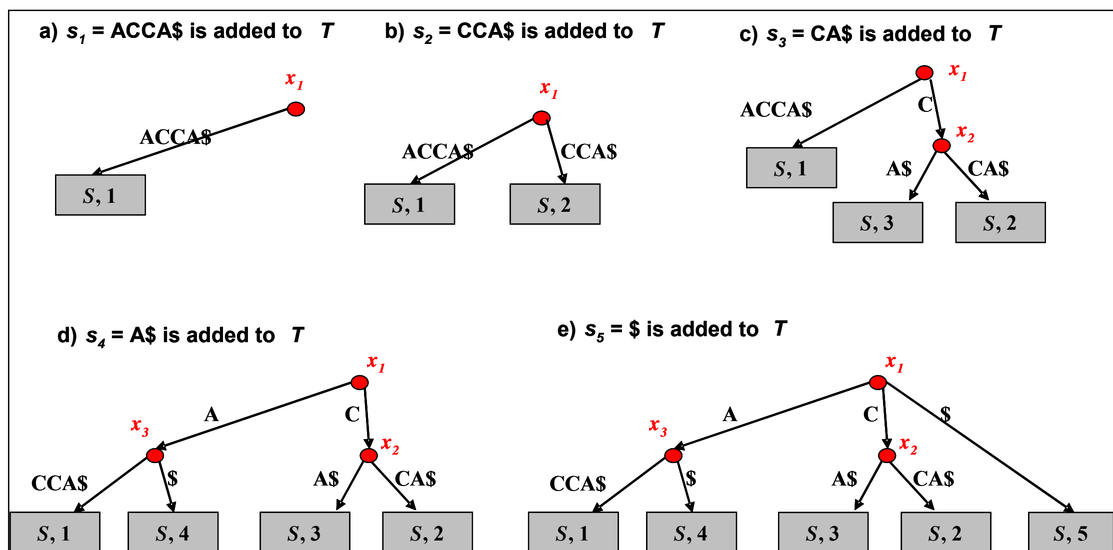


Figure 1-3. Example 1.1 – Naïve construction of the suffix tree T for the string S = ACCA$, which leads to O(|$S$|^2) time.

In the fully constructed tree (Figure 1-3e), there are three branch nodes: the node x_1 is the root of the tree, and the two branch nodes connected to the root are x_2 and x_3. The leaves of the tree are denoted as (S, i), where $1 \leq i \leq 5$.

1.1.2. The space and the time complexity of the algorithms for the suffix tree construction

This naive approach of constructing a suffix tree results in O(l^2) run-time, for a string of length l. However, a suffix tree can be efficiently constructed in O(l) time (Weiner, 1973; McCreight, 1976; Ukkonen, 1995).

The O(l) time construction of suffix trees is based on suffix links. A suffix link connects every non-root branch node to another specifically chosen branch node of the tree. Let T be the suffix tree for a string S. Let S' and S'' be substrings of S, and c a character from the alphabet of S, such that $S' = cS''$. If S' represents a path-label from the root of T to a branch node x', and S'' represents a path-label from the root of T to a branch node x'', then a suffix link connects x' to x'' (Figure 1-4). In addition, suffix links can be used for finding hits with mismatches.

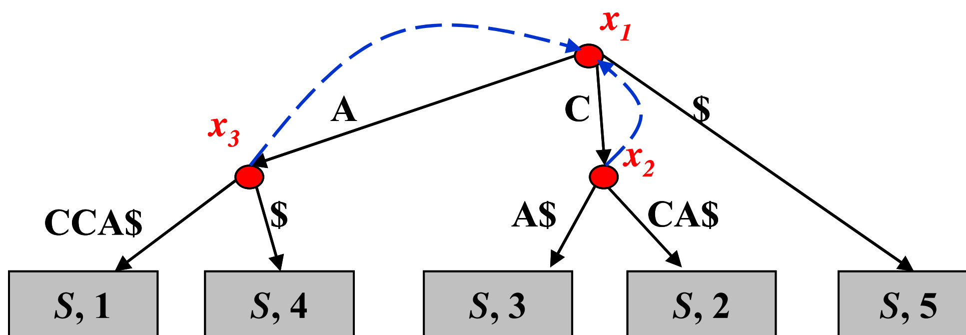


Figure 1-4. The suffix tree *T* for the string *S* = ACCA$, with included suffix links. Suffix links are shown as blue dashed lines connecting the branch nodes x_2 and x_3 to the root of T (x_1).

The important property of suffix trees is that the existence of an arbitrary string P in a suffix tree T can be determined in O(|P|) time. This is done by comparing at most |P| characters of a string P to the characters along the branches of T. The exact matching problem, or finding z occurrences of P in T, can be done in O(|P| + z) time. Let x denote a branch node such that the concatenation of characters from the root of T to x spells out P. Then, all matches of P in T are found by traversing the subtree rooted on x (for example, in depth-first mode), and collecting the starting positions of suffixes corresponding to the leaves in the subtree rooted on x.

Another problem solved by suffix trees is finding the longest common substrings between sequences. Let the generalized suffix tree *T* be constructed for sequences S_1 and S_2 in O($|S_1|$ + $|S_2|$) time. A common substring between these sequences can be found as the concatenation of branch labels from the root of T to a branch node that has the leaves corresponding to both S_1 and S_2 in its subtree. The longest common substring is found by traversing the tree, and storing the information about the longest found substring. In fact, the longest common substring corresponds to the branch node with the deepest string-depth of all branch nodes which are roots of subtrees containing leaves from both S_1 and S_2. In this way, the longest common substring is found in O($|S_1|$ + $|S_2|$) time.

Suffix trees have been widely used in many biological applications (e.g. Kurtz *et al.*, 2001; Höhl *et al.*, 2002; Kurtz *et al.*, 2004; Bray and Pachter, 2004; Apostolico and Denas, 2008). However, the memory requirements of suffix trees are their limiting factor. For example, the best implementations of suffix trees require at least 10 bytes per each character of an input string (sequence), and often more (15-20 bytes) (Kurtz, 1999). Thus, the application of suffix trees to data sets of large genomes becomes demanding (e.g., human genome is $3 * 10^9$ base pairs long). In order to reduce the memory usage of suffix trees, data structures with different trade-offs between the space and the time requirements were proposed (e.g. Hunt, 2003; Ristov, 2003). Among them, a suffix array, introduced by Manber and Myers (1993) emerged as a widely accepted space-efficient alternative to a suffix tree.

1.1.3. Suffix Array

Manber and Myers proposed a data structure called suffix array (1993). Suffix array (SA) is an array of positive integers, corresponding to the starting positions of the lexicographically ordered suffixes of a string.

Example 1.2:

Let *S* = ACCA$. The suffixes of *S* are s_1 = ACCA$, s_2 = CCA$, s_3 = CA$, s_4 = A$, and s_5 = $, and the character $ is a sentinel character as before.

First, the lexicographically ordered list of suffixes of *S* is formed:

s_1 = ACCA$

s_4 = A$

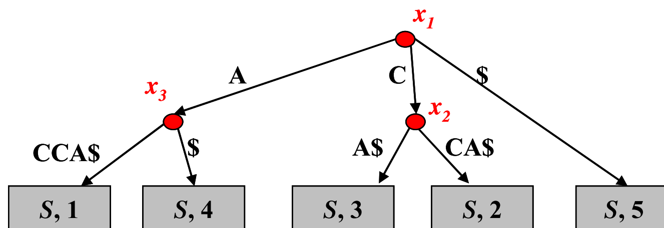
s_3 = CA$

s_2 = CCA$

s_5 = $

Next, the suffix array of *S*, *SA*, is constructed from the list of the lexicographically ordered suffixes of *S*. For example, *SA*[1] = 1, since the first suffix in the lexicographical order is, at the same time, the first suffix of *S* (s_1). *SA*[2] = 4, since the second suffix in the lexicographical order is the fourth suffix of *S* (s_4). Finally, *SA* = [1, 4, 3, 2, 5].

a) The suffix tree for *S* = ACCA$



b) The suffix array for *S* = ACCA$

i	s_i	*SA*[*i*]
1	ACCA$	1
2	A$	4
3	CA$	3
4	CCA$	2
5	$	5

Figure 1-5. The suffix tree and the suffix array for S = ACCA$. The suffix array, *SA*, for *S* can be obtained by preorder traversing of the suffix-tree. For example, *SA*[1] = 1, that is, the first suffix in the lexicographical order, s_1, corresponds to the leaf (*S*, 1), which is the first leaf encountered in the preorder traversal of the leaves of the suffix tree. The second encountered leaf is (*S*, 4) which corresponds to the second suffix in the lexicographical order, s_4, etc.

1.1.4. The space and the time complexity of the algorithms for suffix array construction

The suffix array data structure contains integer values that represent indexes (starting positions) of lexicographically sorted suffixes of a string. The maximal value that can

be stored in the suffix array is l for a string of length l. To store the value l, at least $\lceil \log_2 l \rceil$ bits are needed. Thus, the whole suffix array containing l entries could be stored in $O(l \log_2 l)$ bits. However, in practice, each entry of the suffix array is stored as a 32-bit integer value. Thus, the memory required for the storage of a suffix array is usually $4l$ bytes for the sequence of length l, when l is at most 2^{32}. In some cases, due to the specificities of some suffix array construction algorithms, 4 bytes per input character only cover sequences shorter than 2^{31} characters (Manzini and Ferragina, 2004).

A lightweight algorithm for the construction of the suffix array requires only a small amount of extra space in addition to the space needed for the storage of a string and the accompanying suffix array, i.e., the construction of the suffix array by a lightweight algorithm takes only a little more than $5l$ bytes of space (e.g. Manzini and Ferragina, 2004).

The naive approach to suffix array construction would require $O(l^2 \log l)$ time for a string of length l: sorting l elements of an array by an efficient sort algorithm on average takes $O(l \log l)$ time (e.g. Hoare, 1962; Williams, 1964), and the comparison of two elements (which are suffixes in the case of SA), takes $O(l)$ time. Thus, the overall time needed for the suffix array construction is bounded by $O(l^2 \log l)$. However, a suffix array can be constructed from the corresponding suffix tree in $O(l)$ time by the preorder visit of the suffix tree (see Figure 1-5).

In the last two decades many algorithms for the direct suffix array construction have been proposed. The theoretically best time achieved is $O(l)$ for a string of length l (Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003; Kim *et al*., 2005). These linear time suffix array construction algorithms are all recursive algorithms: a suffix array is constructed for a substring of an observed string in a recursive step. The recursion ends when a sufficiently short suffix is encountered, which is then sorted by a linear procedure (see Puglisi *et al*., 2007)

In the recent study, Puglisi *et al*. (2007) compared the memory requirements and the speed of almost twenty SA construction algorithms on a variety of data sets. Interestingly, they found that the three fastest implementations are based on non-linear algorithms with the $O(l^2 \log l)$ time worst case behavior (Manzini and

Ferragina, 2004; Maniscalco and Puglisi, 2006; Maniscalco and Puglisi, 2007), and not the implementations of the algorithms running in $O(l)$ time. The authors also showed that, at the same time, these three non-linear solutions required in total considerably less space (5 - 6 bytes per character of the input data) than the algorithms running in the theoretically best time. For example, among the SA linear construction algorithms, the implementation of the algorithm of Ko and Aluru (2003) required at least 7 bytes, and the implementation of the algorithm invented by Kärkkäinen and Sanders (2003) required at least 10 bytes per input character.

However, some problems that can be solved by suffix trees cannot be solved by the basic suffix array structure in the same time-complexity. For example, finding a string P in a suffix tree built for the string S takes $O(|P|)$ time, since at most $|P|$ comparisons have to be made along the branches of the suffix tree starting with the branches connected to the root of the suffix tree (Sections 1.1.1 and 1.1.2). In contrast, finding P in a suffix array built for S takes $O(|P| \cdot \log |S|)$ time: The pattern P can be found by the binary search of the lexicographically sorted suffixes of S, where the binary search of the suffix array takes $O(\log|S|)$ time, and each comparison between two suffixes corresponding to the suffix array entries takes $O(|P|)$ time. However, using an additional data structure, the lcp-array (see Section 1.1.5), this problem can be solved in $O(|P| + \log|S|)$ time (Manber and Myers, 1993). Abouelhoda *et al*. (2004) showed that P can be found in S in $O(|P|)$ time using an enhanced suffix array (ESA), the suffix array data structure supplemented with other data structures (Section 1.1.5).

1.1.5. Enhanced Suffix Array

An enhanced suffix array (ESA) is a suffix array accompanied by additional data structures. ESA represents the trade-off between the space and the time requirements of its predecessors, suffix trees and suffix arrays. As explained in the previous section, suffix array requires $4l$ bytes for an input sequence of length l, where l has to be less than 2^{32}. In contrast, the suffix tree data structure requires at least $10l$ bytes of space for the string of length l (Kurtz, 1999; Section 1.1.2). However, the typical string operations (e.g. search for a pattern in a string; Section 1.1.4) using the suffix

array are solved in worse time complexity than the more space-consuming suffix tree data structure.

The memory requirements of an enhanced suffix array are at least $5l$ bytes for a string of length l (Abouelhoda *et al*., 2004). The suffix array requires $4l$ bytes, and the rest comes from the additional data structures: the longest common prefix array or the longest common prefix table (lcp-array or lcp-table), and some other data structures which may be needed for some applications (Abouelhoda *et al*., 2004). The lcp-array contains the integer values representing the length of the longest common prefix between two adjacent suffixes, when suffixes are sorted in the lexicographical order (see Example 1.3). The length of the longest common prefix of two suffixes of the string of length l, takes $\lceil \log_2 l \rceil$ bits, but in practice it is stored as 4-byte integer for strings where l is less than 2^{32}. Depending on the data, the length of the longest common prefix of two adjacent suffixes can be significantly shorter than l, so it can be represented by a one-byte integer, plus some small additional data. Thus, the storage of the lcp-array takes typically between l and $4l$ bytes.

Abouelhoda *et al*. (2004) showed that each algorithm that uses the suffix tree data structure can be adequately replaced by an enhanced suffix array algorithm in the same time complexity. They have introduced the conceptual lcp-interval tree (Figure 1-6). Traversing of the lcp-interval tree simulates the traversal of the corresponding suffix tree, so that every operation requiring the traversal of a suffix tree can be accomplished in the time-complexity using the conceptual lcp-interval tree.

<u>Example 1.3</u>

Let S = ACCA\$. The suffixes of S are s_1 = ACCA\$, s_2 = CCA\$, s_3 = CA\$, s_4 = A\$, and s_5 = \$. The suffix array SA for S has already been constructed in the Example 1.2. Now, the lcp-array $LCPA$ for S is constructed in the following way:

(i) $LCPA[1] = 0$

(ii) $LCPA[i] = lcp(s_{SA[i]}, s_{SA[i-1]})$ for $i = 2, .., |S|$

The value $lcp(s_{SA[i]}, s_{SA[i-1]})$ is the length of the longest common prefix of suffixes $s_{SA[i]}$ and $s_{SA[i-1]}$. The suffix $s_{SA[i]}$ is the suffix of S starting at the position SA[i], and the suffix $s_{SA[i-1]}$ is the suffix of S starting at the position SA[i-1]. Thus, the prerequisite for computing the lcp-array of S is sorting the suffixes of S in lexicographical order. Table 1-1 shows the suffix array and the corresponding lcp-array for S.

Table 1-1. The suffix-array (*SA*) and the lcp-array (*LCPA*) for the string *S* = ACCA$. The right-most column contains lexicographically sorted suffixes of *S*.

i	*SA*[*i*]	*LCPA*[*i*]	Lexicographically ordered suffixes of *S*; *S*[*SA*[*i*]..|*S*|]
1	1	0	ACCA$
2	4	1	A$
3	3	0	CA$
4	2	1	CCA$
5	5	0	$

The next concept is the lcp-interval, denoted lcp-$[a..b]$, $0 \leq a < b \leq l$:

(i) $LCPA[a] < lcp$

(ii) $LCPA[k] \geq lcp$ for all k such that $a + 1 \leq k \leq b$

(iii) $LCP[k] = lcp$ for at least one k such that $a + 1 \leq k \leq b$

(iv) $LCPA[b + 1] < lcp$

For example, the lcp-interval 1-[1..2], represents positions from 1 to 2 in the *LCPA*, which correspond to suffixes $s_{SA[1]}$ = ACCA$, and $s_{SA[2]}$ = A$. The lcp-value of 1-[1..2], which is 1, is the length of the longest common prefix of suffixes $s_{SA[1]}$ = ACCA$ and $s_{SA[2]}$ = A$.

An lcp-interval lcp-$[c..d]$ is said to be embedded in an interval lcp-$[a..b]$, if $a \leq c < d \leq b$. This principle enables the concept of the parent-child relationship between lcp intervals. Here, lcp-$[c..d]$ is considered as a child interval of lcp-$[a..b]$. Next, the virtual lcp-interval tree is constructed using the concept of the parent-child relationship. The root of the lcp-interval tree is the interval 0-[1..|S|], since the length

of the longest common prefix of all suffixes in the tree is zero, and the root interval covers all positions in *S* (Figure 1-6).

The conceptual lcp-interval tree for *S* is built using the enhanced suffix array of *S*: the suffix-array of *S* and the lcp-array of *S*. Every interval of the lcp-interval tree is defined as: *lcp*-[*SA*[*i*].. *SA*[*j*]], where *lcp* is the length of the longest common prefix of suffixes starting at position *SA*[*i*], and ending at position *SA*[*j*] in the list of sorted suffixes. The lcp-interval tree is traversed bottom-up by a linear scan of the lcp array (Abouelhoda *et al.*, 2004).

Figure 1-6a) shows the lcp-interval tree for *S*, and Figure 1-6b) shows the suffix tree constructed for *S*. It is easily seen that the lcp-interval tree has the same topology as the suffix tree without terminal branches and terminal nodes, that is, the suffix tree leaves are not explicitly represented on the lcp-interval tree. Further, the leaves of the lcp-interval tree correspond to the branch nodes of the suffix tree. For example, the lcp-interval tree leaf 1-[1..2] corresponds to the branch node x_3 (Figure 1-6b). Positions from 1 to 2 in the interval 1-[1..2] correspond to the leaves of the suffix tree (*S*, 1) and (*S*, 4). The suffixes of *S* corresponding to leaves (*S*, 1) and (*S*, 4), are the first and the second suffix in the lexicographical order, and $s_1 = s_{SA[1]}$ = ACCA$, and $s_4 = s_{SA[2]}$ = A$.

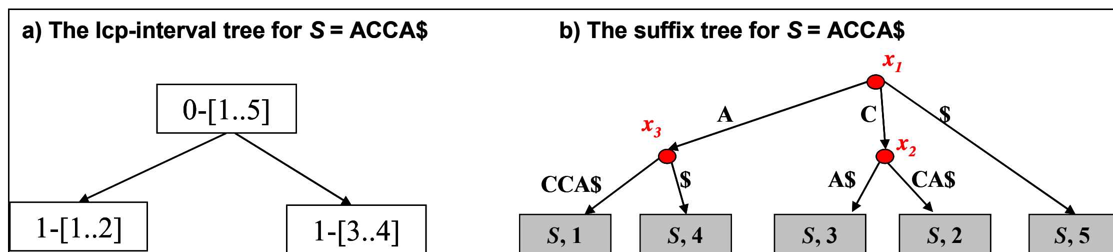


Figure 1-6. The lcp-interval tree and the suffix tree for the string *S* = ACCA$. All non-root nodes of an lcp-interval tree correspond to branch nodes of the analogous suffix tree. The leaf 1-[1..2] on the lcp-interval tree corresponds to the branch node x_3 of the suffix tree, and the leaf 1-[3..4] corresponds to the branch node x_2 of the suffix tree.

The lcp-array can be constructed in O(l) time for a string S of length l from the suffix array of S (Kasai *et al.*, 2001; Manzini, 2004). In addition, the lcp-array can be constructed from the suffix tree of S in O(l) time by the preorder traversal of the suffix tree. This solution is based on the idea that the longest common prefix of two suffixes is the length of the path label of their lowest common ancestor, which can be computed in O(1) time (Harel and Tarjan, 1984; Schieber and Vishkin, 1988). However, the suffix tree has to be preprocessed in O(l) time in order to obtain the length of the path label in O(1) time.

In summary, the theoretically best time-complexity needed for the construction of ESA is O(l), since both the SA and the LCPA for a string S can be constructed in O(l) time. It is the same time complexity needed for the construction of the analogous suffix tree. In addition, the traversal of the lcp-interval tree takes O(l) time (Abouelhoda *et al.*, 2004), which is also the time needed for the traversal of the suffix tree for the same string S of length l. In practice, the best implementations of the suffix array construction are faster then the best implementations of the corresponding suffix tree construction (Puglisi *et al.*, 2007). They also require less memory, since the light-weight SA construction algorithms need only a little extra space besides the SA storage (Puglisi *et al.*, 2007; Section 1.1.4), which is the property of sorting algorithms.

1.1.6. The 64-bit implementation of the lightweight suffix array construction algorithm

In the programs developed as a part of my thesis, *kr* version 2, and *st*, I implemented the suffix tree concept using the more space-efficient data structure, the enhanced suffix array. I used the linear-time algorithm for the construction of the lcp-array (Manzini, 2004), and the suffix array library by Manzini and Ferragina (2004) for the SA construction, since it was one of the fastest and the most space-efficient implementations currently available for the SA construction (Puglisi *et al.*, 2007).

However, the original form of their library limited the analysis to $2^{31} \approx 2\times10^9$ characters. In comparison, the human genome is $3 * 10^9$ base pairs long. To extend the analysis to large genomes, I added a 64-bit implementation of the library. The

current implementation of the 64-bit version requires around 16 bytes per input character, in comparison to 8 bytes per input character of the 32-bit version. This memory requirement comes from the underlying data structures (SA and LCP), which are, in the case of the 32-bit version of the program, based on 4-byte integers, and, in the case of the 64-bit version, on 8-byte integers. This memory drawback of the 64-bit implementation is the trade-off that enables us to work with large data sets. In the current implementation, the 64-bit implementation is intended for the data sets comprising billion or more nucleotides.

1.1.7. Self-index

The memory usage of both suffix trees and (enhanced) suffix arrays is linear with respect to the size of the input data set, although with different constants. However, this still remains a problem for large input data sets. This motivated the invention of data structures that require sublinear storage. This research yielded the concept of the self-index (also compressed self-index, or compressed index), where the text is represented in the compressed form (Navarro and Mäkinen, 2007; Ferragina *et al.*, 2008). The space requirements of self-indexes are proportional to the size of the compressed text.

In the recent study, Ferragina *et al*. (2008) compared implementations of several compressed indexes. They reviewed the representatives of three groups of self-indexes. The first group is the FM-index family, based on the FM-index, the first self-index with space requirements proportional to the k-th order text entropy (Ferragina and Manzini, 2000; Section 1.1.10). The FM-index family relies on the Burrows-Wheeler transform of the text (Burrows and Wheeler, 1994; Section 1.1.8). The second group of indexes comprises compressed suffix arrays (CSA). The original compressed suffix array by Grossi and Vitter (2000) was not a self-index. However, Sadakane proposed a solution (2003) in which he converted the original CSA in a self-index. The third group of self-indexes is the LZ-index group (e.g. Navarro, 2004; Ferragina and Manzini, 2005), based on the Lempel-Ziv compression (Ziv and Lempel, 1978).

In their study, Ferragina and collaborators (2008) observed the behavior of the self-index implementations for two search operations: (*i*) find the number of occurrences of a pattern *P* in the indexed text, and (*ii*) retrieve (locate) all occurrences of *P* in the text. They compared the run-time of four self-index implementations (representatives of the previously mentioned groups), and a plain suffix array. As a result of this study, the compressed indexes were one (in the case of (*i*)) to three orders (in the case of (*ii*)) of magnitude slower than the same operations computed by the suffix array implementation. In contrast, compressed indexes required one order of magnitude less space than the SA (the SA implementation including text required in total 5 bytes per text character). However, the implementations of these compressed indexes required additional space for the construction of the index itself. Explicitly, the representatives of the FM-index family, and the CSA required the construction of the SA prior to the construction of the self-index, and the LZ-index representative required the construction of some auxiliary data structure. In total, the memory usage needed for the construction of self-index was estimated to 5-9 times the text size (Ferragina *et al*., 2008).

Moreover, implementing a self-index is not a trivial task (Ferragina *et al*., 2008). Therefore, only recently these data structures found their way in practical applications. Since several very recent biological applications designed for the efficient short- and long-read[2] alignment to large sequence database (e.g. whole genome) are based on the FM-index (e.g. Langmead *et al*., 2009; Li and Durbin, 2009; Li and Durbin, 2010), it will be described here as a representative of self-indexes (see Sections 1.1.8-1.1.10).

1.1.8. Burrows-Wheeler transform

The Burrows-Wheeler Transform (BWT, or block-sorting compression) is an algorithm that transforms a string into a particular permutation of it. Due to the suitable order of permuted characters, it is also used in the data compression, e.g. bzip2 (Seward, 2007).

[2] In this context, a *read* refers to a DNA sequence of relatively short length. A *short-read* is usually shorter than 100 base pairs, and a long-read is longer than that (e.g. 100 – 10000 base pairs). These short sequences are obtained by sequencing based on the new generation of more efficient techniques.

Let S be a string defined over the alphabet Σ terminated by the character \$, as before (Section 1.1.1). Let SA be the suffix array constructed for a string S.

The Burrows-Wheeler Transform transforms a string S into a string B. B can be obtained from S by rotating S, and then sorting the permutations of S in the lexicographical order. Then, the right-most character of each permutation of S represents a character of B. The complete string B is spelled out by reading the right-most characters of S in top-down direction (Example 1.4; Table 1-2). However, in practice, the BWT of a string is usually computed based on the SA of a string: the BWT of a string S is the string B, such that for i = 1, .., $|S|$ (Example 1.4; Table 1-3):

(i) $B[i] = \$$ for $SA[i] = 1$

(ii) $B[i] = S[SA[i] - 1]$

Example 1.4

Let S = ACCA\$. The suffixes of S are s_1 = ACCA\$, s_2 = CCA\$, s_3 = CA\$, s_4 = A\$, and s_5 = \$. The suffix array for S is SA = [1, 4, 2, 3, 5] (see Example 1.2). Table 1-2 shows the construction of string B = \$CCAA by sorting permutations of S. Table 1-3 represents the BWT of S based on SA (using formulas (i) and (ii)), e.g. $B[1] = \$$; $B[2] = S[SA[2] - 1] = S[4 - 1] = S[3]$ = C, etc.

Table 1-2. The construction of string *B* from the rotations of string *S* = ACCA\$, where *B* is the Burrows-Wheeler transform of *S*.

i	Permutations of *S*	Sorted permutations of *S*	*B*[*i*]
1	ACCA\$	ACCA\$	\$
2	CCA\$A	A\$ACC	C
3	CA\$AC	CA\$AC	C
4	A\$ACC	CCA\$A	A
5	\$ACCA	\$ACCA	A

Table 1-3. The construction of string *B* from string *S* = ACCA$ and the suffix array *SA* of *S*, where *B* is the Burrows-Wheeler transform of *S*.

i	*SA*[*i*]	Lexicographically ordered suffixes of *S*; *S*[*SA*[*i*]..|*S*|]	*B*[*i*]
1	1	ACCA$	$
2	4	A$	C
3	3	CA$	C
4	2	CCA$	A
5	5	$	A

1.1.9. The FM-Index and the backward search algorithm

The FM-index is a self-index data structure based on the suffix array and the BWT of the input data (Ferragina and Manzini, 2000). It will be described here as a representative of index data structures of a new generation, which requires sublinear memory storage at the expense of a somewhat slower run-time needed for typical suffix tree operations (Ferragina *et al*., 2008; Section 1.1.7). Explicitly, the FM-index occupies $O(lH_k(S)) + o(l)$ bits for a string S of length l, where $H_k(S)$ is the k-th order entropy of S (Section 1.1.10). In the comparison of Ferragina *et al*. (2008), two variants of the FM-index required an order of magnitude less space than the suffix array, while the problem of finding pattern occurrences was slower up to the order of magnitude when compared to the plain SA (see Section 1.1.7).

Suffix Array Interval

Let S be a string defined over an alphabet Σ, SA be the suffix array of S, and P a substring of S. All occurrences of P in S are covered by an interval in SA. This can be seen by looking at the lexicographically sorted suffixes of S (for example, see the sorted permutations of S in Table 1-4). The suffix array interval for P is defined as $[L_P, R_P]$ (see Example 1.5):

(i) $L_P = \min \{k: P \text{ is the prefix of } S_{SA[k]}\}$

(ii) $R_P = \max \{k: P \text{ is the prefix of } S_{SA[k]}\}$

In addition, if P is an empty string, then $L_P = 1$, and $R_P = |S|$. The set of all starting positions of P in S is defined as $\{SA[k]: L_P \leq k \leq R_P\}$ (see Example 1.5).

Backward search

Let B be the BWT of S, and let $c \in \Sigma$. The two functions of c are defined: $C(c)$, and $Occ(c, i)$:

(i) $C(c)$ is the number of characters in $S[1..|S|-1]$ which are lexicographically smaller than c (including repetitions of characters)

(ii) $Occ(c, i)$ is the number of occurrences of c in $B[1..i]$, and $i = 1, .., |S|$

If P is a substring of S, then the following property of suffix array intervals holds (Ferragina and Manzini, 2000):

(i) $L_{cP} = C(c) + Occ(c, L_P - 1) + 1$

(ii) $R_{cP} = C(c) + Occ(c, R_P)$

This property enables the *backward search* of P in order to find P in S. The backward search algorithm *BW_Count* (Ferragina and Manzini, 2000) starts with the last character of P, and then goes backwards (Figure 1-7). If P exists in S, the number of occurences of P is returned. Otherwise, 0 is returned. If an *Occ* value is obtained in O(1) time for any $Occ(c, i)$, $i = 0, .., |S|$, then P is found in S in $O(|P|)$ time. Finally, the two main operations on the FM-index are:

(i) counting the number of occurences of P in S (algorithm *BW_Count*; Figure 1-7)

(ii) finding (the positions of) all z occurences of P in S

In comparison, the suffix tree computation of (*i*) takes $O(|P|)$ time (each branch node should store the number of leaves in its subtree), and the computation of (*ii*) is done in $O(|P| + z)$ time (Sections 1.1.1 and 1.1.2).

```
Algorithm BW_Count
Require: C
Require: Occ
Require: P
Ensure: number of occurences of P in S
i = |P| /* position in P */
c = P[|P|] /* last character in P */
L_P = C[c] + 1
R_P = C[cnext] /* cnext is the character following c in Σ */
while ((L_P < R_P) and (i ≥ 2)) do
  c = P[i - 1];
  L_P = C[c] + Occ(c, L_P - 1) + 1
  R_P = C[c] + Occ(c, R_P)
  i = i - 1
if (R_P < L_P) then return 0
else return (R_P - L_P + 1)
```

Figure 1-7. Algorithm *BW_Count* (Ferragina and Manzini, 2000)

Example 1.5

Let S = ACCA\$ as before. The suffixes of S are s_1 = ACCA\$, s_2 = CCA\$, s_3 = CA\$, s_4 = A\$, and s_5 = \$, so the suffix array built for S is SA = [1, 4, 2, 3, 5] (see Example 1.2). Let the substring P = CA be the pattern we wish to search for in S. The suffix-array interval for P is [3, 4], since the substring P occurs in the suffixes s_3 = CA\$ and s_2 = CCA\$ of S, and SA[3] = 3 and SA[4] = 2 (see Table 1-4).

Table 1-4. Finding the suffix array interval for the substring *P* = CA in the string *S* = ACCA$.

I	*SA*[*i*]	Lexicographically ordered suffixes of *S*; *S*[*SA*[*i*]..|*S*|]
1	1	ACCA$
2	4	A$
3	3	**C**A$
4	2	**C**CA$
5	5	$

$L_P = 3, R_P = 4$

The array C for the $c \in \Sigma = \{A, C\}$ is:

C['A'] = 0

C['C'] = 2

The value C['A'] is 0, since no character from S is lexicographically smaller than 'A'. C['C'] is 2, since only 'A' is lexicographically smaller than 'C', and the character 'A' occurs twice in S. Next, the array *Occ* is computed, where $Occ(c, i)$ is the number of occurrences of c in $B[1..i]$, and B = $CCAA (see Table 1-3), e.g.:

Occ('A', 1) = 0, since $B[1..1]$ = $

Occ('A', 2) = 0, since $B[1..2]$ = $C

Occ('C', 1) = 0, since $B[1..1]$ = $

Occ('C', 2) = 1, since $B[1..2]$ = $C

Now, P is searched in S starting from the last character, 'A' (Table 1-5). The number of characters lexicographically preceding 'A' in S is 0, that is, C['A'] = 0. The algorithm starts by initializing $L_P = C['A'] + 1 = 1$, and $R_P = C['C'] = 2$. The next observed character from P is 'C'. The new values of L_P and R_P are computed as:

$L_P = C['C'] + Occ['C', L_P - 1] + 1 = 2 + 0 + 1 = 3$, and

$R_P = C['C'] + Occ['C', R_P] = 2 + 1 = 3$

while $Occ['C', L_P - 1] = Occ['C', 0] = 0$, and $Occ['C', R_P] = Occ['C', 2] = 1$.

Since 'C' is at the first position of P, the search stops here, and the value returned from the function is $R_P - L_P + 1 = 3 - 3 + 1 = 1$. That is, P occurs only once in S.

Table 1-5. Example 1.5 – Backward search of the number of the occurrences of the substring *P* = CA in the string *S*.

i	*P*	*c* = *P*[*i*]	*C*[*P*[*i*]]	L_P	R_P
2	CA	A	0	1	2
1	CA	C	2	3	3

1.1.10. The space and the time-complexity of the FM-index

The FM-index is a *self-index*. A self-index is a data structure built over a string *S* that enables the search of any substring *P* in *S* without requiring the explicit storage of *S*, as *S* can be derived from the self-index. The important property of the FM-index is that its storage requirement is close to the theoretically smallest possible amount, which is the *k*-th order entropy of *S* (Ferragina and Manzini, 2000).

Let $|S| = l$, and let l_i be defined as the number of occurrences of the *i*-th character from Σ. Let *con* be a *k*-length substring (also called context) of *S*. Let S^{con} be a string formed as the concatenation of the characters following *con* anywhere in S taken from the left to the right of *S*. The *k*-th order entropy of *S*, $H_k(S)$, for $k \geq 0$, is defined in the following way:

(i) $$H_0(S) = -\sum_i \frac{l_i}{l} log_2 \frac{l_i}{l}$$

(ii) $$H_k(S) = \frac{1}{l} \sum_{con \in \Sigma^k} \left|S^{con}\right| H_0\left(S^{con}\right)$$

Example 1.6

Let *S* = ACCA$ as before. The alphabet over *S* is Σ = {A, C, G, T}. Let us compute the H_0(S) and H_1(S). First, the number of occurrences of each alphabet letter in *S* determined: $l_1 = 2$, since the first alphabet letter (A) occurs twice in *S*. Similarly, $l_2 = 2$, $l_3 = 0$, and $l_4 = 0$.

Now, the zero order entropy $H_0(S)$ is (using (i)):

$$H_0(S) = -\left(\frac{2}{4}log_2\frac{2}{4}+\frac{2}{4}log_2\frac{2}{4}+0+0\right) = -2\cdot\left(\frac{1}{2}log_2\frac{1}{2}\right) = 1$$

Next, to compute the first order entropy $H_1(S)$, $H_0(con)$ for each context has to be computed. Here, *con* is a 1-length substring of *S*, so in total there are two contexts: con_1 = A and con_2 = C. Further, two subsequences of *S* has to be formed: S^{con1} and S^{con2} which are comprised of characters following *con1* and *con2* in *S*, respectively. Thus: S^{con1} = C$ (the first occurrence of *con1* in *S* is followed by C, and the second occurrence is followed by $), and S^{con2} = CA (the first occurrence of *con2* in *S* is followed by C, and the second occurrence is followed by A). The zero-order entropy of S^{con1} and S^{con2} is $H_0(S^{con1})$ = $H_0(S^{con2})$ = $-2\left(\frac{1}{2}log_2\frac{1}{2}\right) = 1$. Finally, $H_1(S)$ is computed using (ii):

$$H_1(S) = \frac{1}{4}\sum_{con\in\Sigma^1}\left|S^{con}\right|H_0\left(S^{con}\right) = \frac{1}{4}\left(|S^{con1}|H_0\left(S^{con1}\right)+|S^{con2}|H_0\left(S^{con2}\right)\right) =$$

$$= \frac{1}{4}\left(2\cdot 1+2\cdot 1\right) = 1$$

The FM-index occupies $O(lH_k(S)) + o(l)$ (at most $5lH_k(S) + o(l)$) bits for a string *S* of length *l* (Ferragina and Manzini, 2000). In the previous example, the storage required for the FM-index built on *S* = ACCA$ requires $O(l\cdot 1) + o(l) = O(l) + o(l)$ bits.

The existence of a string *P* of length *m* in *S* can be checked in $O(m + \log^{1+\varepsilon} l)$ time, and all *occ* occurences of *P* are found in $O(m + occ\log^{1+\varepsilon} l)$ time, where ε is a small positive constant set in advance ($\varepsilon < 1$) (Ferragina and Manzini, 2000; Ferragina *et al*., 2004).

2. Efficient Estimation of Pairwise Distances between Genomes

2.1. Introduction

Biological sequences are traditionally compared using alignment methods. In an alignment, homologous residues (nucleotides or proteins) are compared. However, finding the optimal multiple sequence alignment is an NP complete problem (Wang and Jiang, 1994). Thus, many alignment tools employ some heuristic to speed up the computation. For example, many popular multiple sequence alignment methods (MSA) (see Chapter 1) are based on the progressive alignment procedure (Notredame, 2007), e.g. ClustalW (Larkin *et al*., 2007), Muscle (Edgar, 2004), MAFFT (Katoh and Hiroyuki, 2008), ProbCons (Do *et al*., 2005), and T-Coffee (Notredame *et al*., 2000). The progressive technique works in the following way: the MSA is constructed by combining the pair-wise alignments between sequences; first, the more closely related sequences are added to the MSA, and then proceeding to the more distant pairs. In the initial step, the order of the sequences added to the MSA is based on the guide tree (which is a phylogenetic tree of sequences) constructed by some other methods. The final results of the progressive alignment procedure are both the MSA and the phylogenetic tree of input sequences.

Recently, Edgar and Batzoglou (2006) compared popular multiple-sequence alignment tools. Among the evaluated programs, MAFFT (Katoh and Hiroyuki, 2008), and MUSCLE (Edgar, 2004) showed a good trade-off between computational requirements and accuracy: they were the most efficient programs for the data sets with large number (100 or more) of sequences. In general, they were faster and more accurate than the popular ClustalW (Larkin *et al*., 2007). In comparison, the most accurate program in the study, ProbCons (Do *et al*., 2005), did not scale well for large data sets. Among other tools that were not included in the study, the efficient program MAVID (Bray and Pachter, 2004) is widely-used for the comparison of many short syntenic sequences (e.g. White *et al*., 2009). Another group of alignment-tools are pair-wise sequence alignment tools, which are more efficient, but generally less accurate than MSA. The representative of this group is the popular program MUMmer, developed for the comparison of large genomes (Kurtz *et al*., 2004).

However, the alignment of data sets consisting of many large complete genomes (e.g. the human genomes consists of 3.2 billion base pairs) becomes difficult to compute. As a more efficient option, alignment-free methods were proposed for the computation of distances between genomes, or for finding similar regions between them. In addition, Höhl *et al*. (2006) showed that on rearranged input sequences alignment-free methods produced more accurate phylogenetic trees than the global alignment methods.

The alignment-free methods of distance estimation were mostly developed in two directions (Vinga and Almeida, 2003). The first group encompasses methods where a pair-wise distance is estimated from word frequencies of compared sequences (e.g. Stuart and Berry, 2003; Sims *et al*., 2009). The second group is based on ideas derived from information theory (e.g. Chen *et al*., 2000; Li *et al*., 2001; Otu and Sayood, 2003; Ulitsky *et al*., 2006). However, the alignment-free methods developed for the estimation of evolutionary distances are generally not based on an evolutionary model. Thus, the distances produced by these methods do not scale linearly with the substitution rate, the standard measure of evolutionary distance used by biologists (see Chapter 1).

We have recently developed an alignment-free pair-wise distance measure for closely related DNA sequences, K_r, which is based on an evolutionary model (Haubold *et al*., 2009). K_r distances gave the most accurate phylogenetic results when compared to other recently introduced alignment-free distance measures (Haubold *et al*., 2009). However, in the original implementation of K_r (*kr* version 1), a suffix tree was constructed and traversed for each pair of sequences. Thus, all pair-wise distances of n input sequences were computed from $n(n-1)$ suffix trees. The run time of this implementation was slow for large sequence samples.

Here, I address the problem of the efficient computation of all K_r pair-wise distances from n input sequences. I found an algorithmic solution to this problem and implemented it in the program *kr* version 2. This program is scalable to large data sets of complete genomes (Domazet-Lošo and Haubold, 2009). The new algorithm enables the extraction of all $\binom{n}{2}$ pair-wise K_r distances from a single traversal of a single suffix tree of n sequences (Section 2.2.5). Thus, the time needed for the suffix

tree construction phase was reduced from $O(n^2l)$ to $O(nl)$. The efficiency and the scalability of the new implementation were shown on both simulated and real data sets of complete genomes (Sections 2.3 and 2.4), which included the 825 genomes of HIV-1 strains, 13 genomes of enterobacteria, and the complete genomes of 12 *Drosophila* species (Domazet-Lošo and Haubold, 2009).

2.2. Methods

2.2.1. Definition of an alignment-free estimator of the rate of substitution, K_r

Let $\boldsymbol{S} = \{S_1, .., S_n\}$ be a set of n (closely related) nucleotide sequences represented as strings over the alphabet {A, C, G, T}. Each sequence is represented by its forward and reverse strand, and terminated by a unique character, which is not a member of the alphabet.

Let (S_i, S_j) be a pair of sequences from $\boldsymbol{S}$. Let $h_{i,j,p}$ denote the shortest prefix of a suffix of $S_i[p .. |S_i|]$, $1 \leq p \leq |S_i|$, that is absent from S_j. This prefix is called *shortest unique substring*, or *shustring* (Haubold *et al*., 2005; Haubold and Wiehe, 2006). The value $o_{i,j}$, *observed average shustring length*, is defined as the sum of all shustring lengths for a pair (S_i, S_j) over all positions $p = 1,..,|S_i|$ divided by $|S_i|$:

$$o_{i,j} = \frac{1}{|S_i|} \sum_{p=1}^{|S_i|} |h_{i,j,p}| \qquad (2\text{-}1)$$

In general, $o_{i,j} \neq o_{j,i}$. This asymmetry is mostly caused by local similarity, or similarity to the repetitive element (Haubold *et al*., 2009; Haubold *et al*., 2008):

Let $|S_i| < |S_j|$, and S_i and S_j be similar along the positions of S_i (Figure 2-1a). In this case, $o_{i,j} > o_{j,i}$, since only the part of S_j that is similar to S_i can have observed shustring lengths greater than the values obtained by chance alone. The remaining positions of S_j have the observed shustring lengths as obtained by chance. On the other hand, the observed shustring length for every position in S_i is expected to be greater than the value obtained by chance. On average, the value of observed shustring lengths should be smaller for S_j than for S_i. This problem can be mitigated

by excluding short shustrings (i.e. shorter than by chance alone) from the computation of the observed average shustring length between a sequence pair.

Let S_i and S_j be sequences of the same length, and let a part of S_j be similar to the repetitive element of S_i (Figure 2-1b). Again, along the similar positions, the observed shustring lengths will be greater than the shustring lengths obtained by chance alone. The sequence S_i has more copies of the matching subsequence, i.e., more positions for which the observed shustring lengths will be greater than the shustring lengths obtained by chance alone, so , $o_{i,j} > o_{j,i}$.

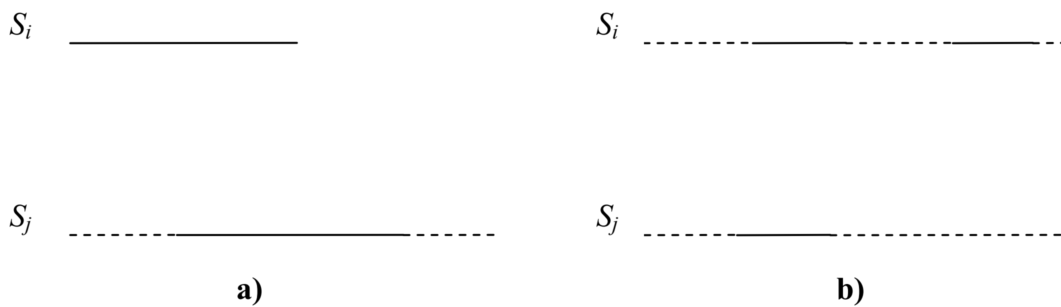


Figure 2-1. Two main causes of the asymmetric values of the observed average shustring length $o_{i,j}$ and $o_{j,i}$ between S_i and S_j. Similar parts are denoted with solid line, and parts which are not similar are denoted with dashed line. In a), S_i is locally similar to S_j, so $o_{i,j} > o_{j,i}$. In b), S_j has only one copy of a similar part, and S_i has two copies, and again, $o_{i,j} > o_{j,i}$.

In general, the number of duplications increases the average observed shustring length faster than mutation decreases it. Hence, to keep the symmetry of the observed average shustring length, the smaller of the values $o_{i,j}$ and $o_{j,i}$ is used as the observed average shustring lengths for both (S_i, S_j) and (S_j, S_i):

$$o_{i,j} = o_{j,i} = \min \{o_{i,j}, o_{j,i}\} \qquad (2\text{-}2)$$

Further, the observed average shustring length is used for the approximation of the expected shustring length, *shulen*. The expected shustring length of a pair (S_i, S_j) is then used to determine $d_{i,j}$, the number of pair-wise mismatches per nucleotide between S_i and S_j since they diverged from their last common ancestor (derived in

Haubold *et al.*, 2009). Finally, the number of nucleotide substitutions between S_i and S_j is estimated by Jukes-Cantor equation (Jukes and Cantor, 1969):

$$K_r(S_i, S_j) = -\frac{3}{4} ln(1 - \frac{4}{3} d_{i,j}) \quad (2\text{-}3)$$

2.2.2. Problem statement

Let $\boldsymbol{S}$ be a set of n nucleotide sequences $\boldsymbol{S} = \{S_1, .., S_n\}$. Without loss of generality, let l denote the length of each sequence. Thus, the total length of n sequences is nl. The distance matrix K_r contains all pairwise evolutionary distances between sequences from $\boldsymbol{S}$ which are based on the K_r measure that is, $K_r^{i,j} = K_r(S_i, S_j)$ for every pair (S_i, S_j) ($i = 1, .., n, j = 1, .., n$). The elements $K_r^{i,i}$ are equal to zero, since they represent the distance from a sequence S_i to itself. The observed average shustring length $o_{i,j}$ is used to estimate $K_r^{i,j}$ (see Sections 2.2.1, equations (2-1)-(2-3)). The problem is: How to efficiently compute all values $o_{i,j}$ for every pair of sequences (S_i, S_j)?

In the previous approach, implemented in *kr* 1 (Haubold *et al.*, 2009), each value $o_{i,j}$ was determined by constructing and traversing a generalized suffix tree for every pair (S_i, S_j). Here, I propose a new algorithm for the efficient computation of all values $o_{i,j}$, and consequently all $\binom{n}{2}$ values $K_r^{i,j}$ in a single traversal of a generalized suffix tree of n sequences. In the following section, as the motivation for the new approach, I discuss the time complexity of both approaches with respect to the improvement in the time-complexity, and consequently, in the speed (Section 2.3.5), of the new approach.

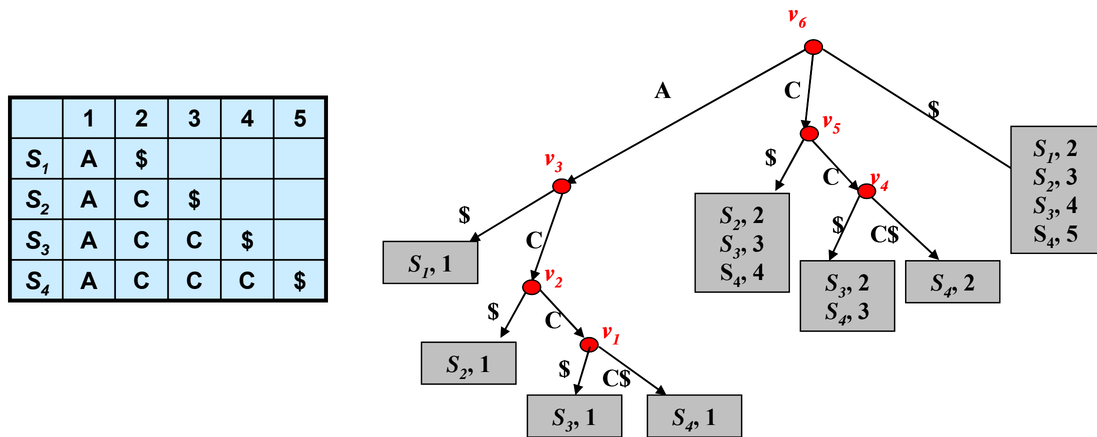
2.2.3. Time complexity analysis of the previous approach (*kr* 1)

Let (S_i, S_j) be a pair of sequences from $\boldsymbol{S}$, $\boldsymbol{S} = \{S_1, .., S_n\}$. Sequence S_i is referred to as *query* sequence and sequence S_j is referred to as *subject* sequence. Conversely, for a pair (S_j, S_i), S_j is referred to as *query* sequence and S_i is referred to as *subject* sequence. In the approach used in *kr* 1, a generalized suffix tree T was constructed

for each pair (S_i, S_j) (Haubold *et al.*, 2009). Thus, for the computation of all K_r values $n(n\text{-}1)$ trees were constructed and traversed.

For a pair of sequences (S_i, S_j), with the total sequence length $2l$, both the suffix tree construction and the suffix tree traversal takes O(l) time. Thus, in the approach used in *kr* 1, the time complexity of the construction of $n(n\text{-}1)$ suffix trees for all pairs from $\boldsymbol{S}$ is O(n^2l).

Let the suffix $S_i[p..|S_i|]$ correspond to a terminal node (S_i, p). Let the suffix $S_j[r..|S_j|]$ correspond to a terminal node (S_j, r), which has the longest common prefix with the suffix $S_i[p..|S_i|]$ when compared to all other suffixes of S_j. The shustring length $|h_{i,j,p}|$ is determined when the branch node v of T is visited, where v is the lowest common ancestor of (S_i, p) and (S_j, r) (Figure 2-2). Then, the value $|h_{i,j,p}|$ is added to the sum $o_{i,j}$ in O(1) time.



	1	2	3	4	5
S_1	A	$			
S_2	A	C	$		
S_3	A	C	C	$	
S_4	A	C	C	C	$

Figure 2-2. A generalized suffix tree of sequences from the set *S* = {*S*₁, *S*₂, *S*₃, *S*₄}. Each branch node is denoted as a red circle, and each leaf as a gray rectangle. Each suffix corresponds to a terminal node of the suffix tree. For example, $S_3[1..|S_3|]$ = ACC$ corresponds to the terminal node (S_3, 1). The shustring of the suffix $S_3[1..|S_3|]$ with respect to S_4 is $h_{3,4,1}$ = ACC$, since the longest common prefix of $S_3[1..|S_3|]$ and any suffix of S_4 is the common prefix of $S_3[1..|S_3|]$ and $S_4[1..|S_4|]$. The shustring $h_{3,4,1}$ is determined as the path-label from the root (v_6) to the branch node v_1 (which is the lowest common ancestor of (S_3, 1) and (S_4, 1)) plus the first character on the label of the branch connecting v_1 to (S_3, 1).

There are l positions in S_i, which correspond to l terminal nodes in T. The addition of all $|h_{i,j,p}|$ values for S_i takes $O(l)$ time. Thus, the number of additions during the traversal of all $n(n-1)$ trees is $n(n-1)l$. Hence, the overall time needed for the traversal of $n(n-1)$ suffix trees with the computation of K_r values, is $O(n^2l)$ (see Table 2-1).

2.2.4. Time complexity analysis of the new approach (*kr* 2)

Let $\boldsymbol{S}$ be a set of n nucleotide sequences $\boldsymbol{S} = \{S_1, .., S_n\}$ of length l as before. In the new approach, implemented in *kr* 2, a single generalized suffix tree for all sequences from $\boldsymbol{S}$ is constructed (Domazet-Lošo and Haubold, 2009). In this approach, every sequence S_i ($i = 1, .., n$) is regarded both as a query sequence for a pair (S_i, S_j), and, at the same time, as a subject for n-1 pairs (S_j, S_i) ($j = 1,..,n$ and $i \neq j$). The suffix tree construction phase of this approach takes $O(nl)$ time, since the total length of n sequences is nl.

The bottom-up traversal of the suffix tree with the computation of all $o_{i,j}$ is the same as in the first approach, $O(n^2l)$. This upper bound is determined in the following way: the traversal of a suffix tree of n sequences is proportional to the number of terminal nodes in the tree, nl. For a suffix $S_i[p .. |S_i|]$, and a subject sequence S_j, the value $|h_{i,j,p}|$ is added to $o_{i,j}$ with the time complexity $O(1)$. Since there are n-1 subjects, the number of additions[3] for a suffix $S_i[p .. |S_i|]$ is, at most, n-1. Notice that the number of additions in this new approach can be less than n-1 per suffix, as discussed later (see Section 2.2.5).

For all l suffixes of S_i and every S_j ($i, j = 1,..,n$, $i \neq j$), the number of $|h_{i,j,p}|$ values added to $o_{i,j}$ is $(n-1)l$. Since there are n sequences that can be considered as query sequence, S_i, the traversal of a generalized suffix tree with computation of all K_r values takes $O(n^2l)$ time.

---

[3] If we could determine all values $|h_{i,j,p}|$ (with respect to every sequence S_j) when visiting a parent node of a leaf (S_i, p), then for every (S_i, p) the number of additions would be $|\boldsymbol{S} \setminus \{S_i\}|$, that is, n-1. Since the number of terminal nodes in the suffix tree is nl, the upper bound of all addition operations for all calls of this function during the tree traversal is $nl(n-1)$, and therefore, the time complexity of the function is $O(n^2l)$.

In summary, the time complexity analysis of the conceptual model of the previous approach (implemented in *kr* 1), and the new approach (implemented in *kr* 2) is presented in Table 2-1.

Table 2-1. Time complexity of the conceptual models underlying *kr* 1 and *kr* 2

	Previous Approach (*kr* 1)	**New Approach (*kr* 2)**
Suffix Tree Construction	$O(n^2l)$	$O(nl)$
Suffix Tree Traversal	$O(n^2l)$	$O(n^2l)$

2.2.5. Algorithm 1: Computation of all K_r values during the traversal of a generalized suffix tree of *n* sequences

Again, let $\boldsymbol{S} = \{S_1, .., S_n\}$, and let l denote sequence length. Every sequence is represented by its forward and reverse strand. Let T be the generalized suffix tree of $\boldsymbol{S}$. As previously explained (Section 2.2.2), the task is to efficiently find $K_r^{i,j}$ values for every pair (S_i, S_j), $i = 1,.., n$, $j = 1,..,n$.

Here, a new algorithm, Algorithm 1, is described that enables the computation of all $\binom{n}{2}$ $K_r^{i,j}$ values in a single traversal of a generalized suffix tree of n sequences (Domazet-Lošo and Haubold, 2009). Algorithm 1 is shown in Figure 2-3. Example 2-1 illustrates Algorithm 1, and is shown in Figure 2-4.

As detailed in Section 2.2.1, an estimate of the substitution rate S_i and S_j, $K_r^{i,j}$, is a function of $o_{i,j}$, where $o_{i,j} = \frac{1}{|S_i|}\sum_{p=1}^{|S_i|}|h_{i,j,p}|$. Algorithm 1 enables the computation of all $o_{i,j}$ values in a single bottom-up traversal of a generalized suffix tree of n sequences.

Let v denote a branch node of T. In Algorithm 1, every v contains the following six fields:

1. *seqId* – the set of sequence identifiers $\{S_i \mid \exists\ (S_i, p)$ that is a <u>terminal node in the subtree rooted on v</u>[4]$\}$
2. *termSeqId* – the set of sequence identifiers $\{S_i \mid \exists\ (S_i, p)$ that is a <u>terminal child of v</u>$\}$; $v.termSeqId \subseteq v.seqId$
3. *branchChildren* – the set of branch nodes which are children of v
4. *countTermSubtree*$[i]$ - the number of <u>terminal nodes in the subtree rooted on v</u> that refer to S_i
5. *countTermChildren*$[i]$ – the number of terminal children of v that refer to S_i
6. *stringDepth* – the length of the path label of v; the string depth of v represents the length of the longest common prefix of all suffixes corresponding to the terminal nodes in the subtree rooted on v

Let a suffix $S_i[p\ ..\ |S_i|$ correspond to a terminal node (S_i, p), which is a terminal child of a node v. Algorithm 1 is based on following properties:

(i) If there is no terminal node in a subtree rooted on v that corresponds to a suffix of S_j, then the value $|h_{i,j,p}|$ for some S_j cannot be determined when v is visited during the tree traversal. In that case, the counter $v.countTermChildren[i]$ is increased by 1. In general, the number of shustrings that cannot be determined for $S_i \in v.seqId$ is:

$$v.countTermSubtree[i] = \sum_{w_k \in v.branchChildren} w_k.countTermSubtree[i] + v.countTermChilden[i]$$

(ii) If $v.seqId \subset \boldsymbol{S}$, then values $|h_{i,j,p}|$ cannot be determined for $S_j \in \boldsymbol{S} \setminus v.seqId$. Let w be a branch node that is the lowest common ancestor of $S_i[p\ ..\ |S_i|]$ and at least one suffix of S_j ($S_j \in w.seqId$). When w is encountered during the traversal of T, $|h_{i,j,p}|$ is determined, and added to $s_{i,j}$. This is done by applying the function *scanBran* to w.

(iii) All values $|h_{i,j,p}|$ for every S_j ($j = 1, .., n$ and $i \neq j$) are determined, when a node w is encountered during the traversal of T, which is the lowest common

[4] Terminal nodes in the subtree rooted on v also include terminal children of v.

ancestor of $S_i[p..|S_i|]$ and at least one suffix of S_j for every S_j. In other words, $w.seqId = \boldsymbol{S}$.

Let v denote a branch node of T. Algorithm 1 consists of two functions that incorporate properties (i)-(iii):

(i) function *scanTerm* looks up the terminal children of v

(ii) function *scanBran* looks up the terminal nodes in the subtrees rooted on the branch children of v

Let (S_i, p) be a terminal node of v. Then, $|h_{i,j,p}|$ is determined in the following way (4):

(i) if S_j ($S_j \neq S_i$) is referred to by a terminal node in the subtree rooted on v, i.e. $S_j \in v.seqId$, then $h_{i,j,p}$ is the substring that represents the path label to v plus the first character on the branch label from v to (S_i, p). Hence, $|h_{i,j,p}| = v.stringDepth + 1$, and the value of $|h_{i,j,p}|$ is added to $s_{i,j}$ by calling *scanTerm*.

(ii) if S_j ($S_j \neq S_i$) is not referred to by any terminal node in the subtree rooted on v, i.e. $S_j \notin v.seqId$, then $h_{i,j,p}$ cannot be determined when v is visited. However, $|h_{i,j,p}|$ is determined and added to $s_{i,j}$ when a node w is visited, where $S_j \in w.seqId$. A node w can be an immediate parent of v, or some other node which contains v in its subtree, and is visited at some point later during the traversal of T. Thus, $|h_{i,j,p}| = w.stringDepth + 1$, and the value $|h_{i,j,p}|$ is added to $s_{i,j}$ by applying *scanBran* to w. The important property of *scanBran* is that all $|h_{i,j,p}|$ which were not previously added to $s_{i,j}$ by applying *scanTerm* are added at once using the multiplication rule of function *scanBran* (line 22). This principle speeds up the computation of $s_{i,j}$ values.

```
Algorithm 1 Estimate substitution rate
Require: T {suffix tree of n DNA sequences S1,S2,...,Sn}
Require: L {Li =|Si|}
Ensure: Kr {n × n matrix of substitution rates}
 1: for all 1 ≤ i, j ≤ n do
 2:   sij ← 0 {initialize s - pairwise sums of shustring len.}
 3: traverse(root(T),1)
 4: for all 2 ≤ i ≤ n do
 5:   for all 1 ≤ j < i do
 6:     Kr^ij ← kr(min(sij/Li, sji/Lj))

 7: function traverse(v, s)
 8: for all w ∈ v.branchChildren do
 9:   traverse(w, s)
10: scanTerm(w, s)
11: scanBran(w, s)
12: end function

13: function scanTerm(v, s)
14: for all i ∈ v.termSeqIds do
15:   for all j ∈ v.seqIds \ {i} do
16:     sij ← sij + (v.stringDepth + 1)× v.countTermChildren[i]
17: end function

18: function scanBran(v, s)
19: for all w ∈ v.branchChildren do
20:   for all j ∈ v.seqIds \ w.seqIds do
21:     for all i ∈ w.seqIds
22:       sij ← sij +(v.stringDepth + 1)× w.countTermSubtree[i]
23: end function
```

Figure 2-3. Algorithm 1. Computing the *n*-by-*n* matrix K_r for a set of *n* sequences $\{S_1, .., S_n\}$, where $K_r^{i,j}$ is an estimate of the substitution rate between S_i and S_j (Domazet-Lošo and Haubold, 2009).

Example 2.1

Let T be the generalized suffix tree for $\boldsymbol{S} = \{S_1, S_2, S_3, S_4\}$; S_1 = A$, S_2 = AC$, S_3 = ACC$, and S_4 = ACCC$, as shown in Figure 2-4a. As before, the sentinel character of each sequence differs from the other sentinel characters, and is not member of the alphabet.

A suffix $S_i[p .. |S_i|]$ is looked up in T by concatenating the branch labels from the root of T to a terminal node designated (S_i, p). Let (S_i, p) be a terminal child of v_i. The sequence S_i it refers to is considered as *query* when compared to all other *subject* sequences S_j ($S_j \neq S_i$). The string depth of each node v_i is the path length of v_i.

For example, in Figure 2-4a, $S_3[1 .. |S_3|]$ = ACC$ refers to the terminal node (S_3, 1), which is a terminal child of v_1. The path label of v_1 is ACC, and the string depth of v_1 is 3. The sequence S_3 is *query*, and S_1, S_2, and S_4 are *subjects* in the sequence pairs (S_3, S_1), (S_3, S_2), and (S_3, S_4).

Algorithm 1 starts at the root of T by calling the function *traverse*. This function ensures that T is traversed bottom-up, and during the traversal, values $|h_{i,j,p}|$ are determined, and added to the matrix s.

For example, v_1 has two terminal children: (S_3, 1), and (S_4, 1), and the string depth of v_1 is 3. The list of sequences referred to by the terminal nodes of v_1 is: v_1.*termSeqId* = $\{S_3, S_4\}$. Since, there are no other terminal nodes in the subtree rooted on v, v_1.*seqId*= v_1.*termSeqId* = $\{S_3, S_4\}$. The values of the remaining fields are: v_1.*branchChildren* = 0, v_1.*countTermSubtree*[3] = 1, and v_1.*countTermSubtree*[4] = 1.

The shustrings $h_{3,4,1}$ and $h_{4,3,1}$ can be determined when v_1 is encountered during the traversal of T. When determining $h_{3,4,1}$ the sequence S_3 serves as query, and S_4 as subject, and when determining $h_{4,3,1}$, S_4 is query, and S_3 subject.

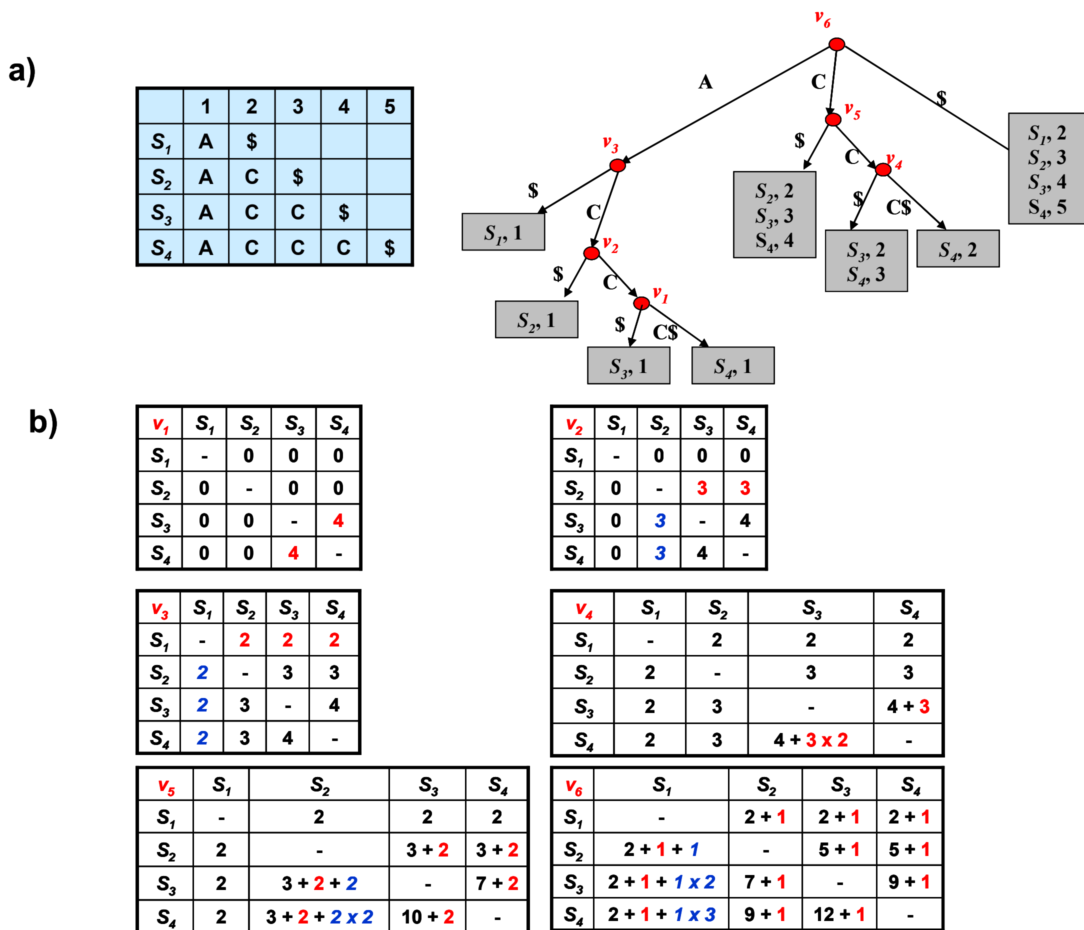
The shustring $h_{3,4,1}$ is ACC$, obtained by the concatenation of the path label of v_1 plus the first character on the branch connecting v_1 and (S_3, 1). The shustring $h_{4,3,1}$ is ACCC, obtained as the concatenation of the path label of v_1 plus the first character on the branch connecting v_1 and (S_4, 1). Here, the length of both shustrings $h_{3,4,1}$ and $h_{4,3,1}$ is $|h_{3,4,1}|=|h_{4,3,1}|=3+1=4$, and these values are added to $s_{3,4}$ and $s_{4,3}$, respectively, by calling function *scanTerm* for v_1.

In general, the shustring length $|h_{i,j,p}|$ for a subject S_j cannot be determined, until S_j is included in *v.seqId*. Thus, the values $h_{3,1,1}$ and $h_{3,2,1}$, and $h_{4,1,1}$ and $h_{4,2,1}$ cannot be determined, since subjects S_1 and S_2 are not elements of v_1*.seqId* (there are no terminal nodes in the subtree rooted on v_1 that refer to S_1 and S_2). Thus, the number of suffixes of S_3 and S_4 for which the shustring length cannot be determined for some S_j (here, S_1 and S_2) is set to 1, or:

$v_1.countTermSubtree[3] = v_1.countTermSubtree[4] = 1$

However, $h_{3,1,1}$ and $h_{4,1,1}$ can be determined when the first node is visited that is the lowest common ancestor of v_1 and has a terminal node referring to S_1 in its subtree. This is v_3, so $|h_{3,1,1}|=|h_{4,1,1}|=1+1=2$. Similarly, $h_{3,2,1}$ and $h_{4,2,1}$ are determined when v_2 is visited during the traversal of T, thus $|h_{3,2,1}| = |h_{4,2,1}| = 2 + 1 = 3$. These values are added to $s_{3,1}$, $s_{4,1}$, $s_{3,2}$ and $s_{4,2}$ by calling function *scanBranch* for v_3 and v_2 respectively.

The states of elements of s during the traversal of T are shown in Figure 2-4b. Finally, after T has been traversed, values $s_{i,j}$ are converted to $K_r^{i,j}$ values (Algorithm 1, lines 4 - 6).



	1	2	3	4	5
S_1	A	$			
S_2	A	C	$		
S_3	A	C	C	$	
S_4	A	C	C	C	$

v_1	S_1	S_2	S_3	S_4
S_1	-	0	0	0
S_2	0	-	0	0
S_3	0	0	-	4
S_4	0	0	4	-

v_2	S_1	S_2	S_3	S_4
S_1	-	0	0	0
S_2	0	-	3	3
S_3	0	3	-	4
S_4	0	3	4	-

v_3	S_1	S_2	S_3	S_4
S_1	-	2	2	2
S_2	2	-	3	3
S_3	2	3	-	4
S_4	2	3	4	-

v_4	S_1	S_2	S_3	S_4
S_1	-	2	2	2
S_2	2	-	3	3
S_3	2	3	-	4 + 3
S_4	2	3	4 + 3 x 2	-

v_5	S_1	S_2	S_3	S_4
S_1	-	2	2	2
S_2	2	-	3 + 2	3 + 2
S_3	2	3 + 2 + 2	-	7 + 2
S_4	2	3 + 2 + 2 x 2	10 + 2	-

v_6	S_1	S_2	S_3	S_4
S_1	-	2 + 1	2 + 1	2 + 1
S_2	2 + 1 + 1	-	5 + 1	5 + 1
S_3	2 + 1 + 1 x 2	7 + 1	-	9 + 1
S_4	2 + 1 + 1 x 3	9 + 1	12 + 1	-

Figure 2-4. Algorithm 1 – Example 2-1. a) Generalized suffix tree T is built from the set of four sequences $\{S_1, S_2, S_3, S_4\}$ listed in the top-left corner of the figure. Branch nodes are shown in red, and terminal nodes as gray rectangles designated as (S_i, p). A terminal node (S_i, p) corresponds to a suffix $S_i[p \,..\, S_i]$. **b)** State of the matrix s after the traversal of each of the branch nodes of T $\{v_1, v_2, v_3, v_4, v_5, v_6\}$. Algorithm 1 starts by calling function traverse for the root node (v_6). Each branch node v_i is visited during the bottom-up traversal of T. Function scanTerm looks up the terminal children of every v_i, and function scanBran looks up the branch children of v_i.

2.2.6. The implementation of *kr* version 2

Algorithm 1 is implemented in the program *kr* version 2 (Domazet-Lošo and Haubold, 2009), which replaces its predecessor, *kr* version 1 (Haubold et al., 2009). In Sections 2.3 and 2.4 the speed gain of the new version, *kr* 2, is demonstrated on both simulated and real data sets.

Apart from *kr* 2 being the faster program, both program versions produce identical results. Moreover, they implement the underlying suffix tree as an enhanced suffix array (Abouelhoda *et al*., 2004), where the suffix array was constructed using the suffix array library of Manzini and Ferragina (2004). However, the original implementation by Manzini and Ferragina was designed for data sets containing up to 2^{31} characters (approximately 2 billion characters, which is less than one strand of the human genome). I extended the usage of the library to larger data sets by implementing its 64-bit version.

The program *kr* version 2 was written in the C programming language and is intended for use under the UNIX environment. It can be used either as a 32-bit or as a 64-bit program, depending on the size of the data set. In particular, the peak memory-usage of the 32-bit version of *kr* 2 is ≈ 9 times the size of the data set, and for the 64-bit version of *kr* 2, it is 17-18 times the size of the data set. This memory requirement comes from the underlying data structures (i.e. enhanced suffix array, and some additional data) which are, in the case of the 32-bit version of the program, based on 4-byte integers, and, in the case of the 64-bit version, on 8-byte integers. The program source and the documentation are available from the web site: http://guanine.evolbio.mpg.de/kr2/

2.3. Analysis of K_r on simulated data sets

2.3.1. Auxiliary programs

For the simulation of samples used in the following analyses the programs *ms* (Hudson, 2002), and *ms2dna* (Haubold and Pfaffelhuber, 2008) were used. The *ms* program generates haplotype samples with the predefined number of single nucleotide polymorphisms (SNPs) per site, which are then converted to a set of nucleotide sequences by *ms2dna*. A *single-nucleotide polymorphism* (SNP) is a variation in a nucleotide sequence between individuals of a species, i.e. different individuals can have a different nucleotide residing at the homologous sequence position. For example, a genome of an individual can contain a nucleotide fragment A**C**CTA, and a genome of another individual of the same species can contain a fragment A**A**CTA at the same positions. These different forms of a gene residing at

the same positions in the sequence are called *alleles*. In this context, a *haplotype* can be considered a set of neighboring SNPs.

The distance matrix correlation coefficient between a K_r distance matrix and a distance matrix obtained from the corresponding alignment was computed using Mantel's test (Mantel, 1967) implemented in the program *zt* (Bonnet and Van de Peer, 2002).

2.3.2. Consistency of K_r

K_r has already been shown to be accurate for long sequence pairs separated by no more than 0.5 substitutions per site (Haubold *et al*., 2009). In this section, a more detailed correlation analysis between K_r and the true substitution rate (obtained from the alignment-based substitution rate) is presented. The analysis was based on samples of 10 homologous sequences of different lengths affected by a variable number of single nucleotide polymorphisms (SNPs) per site, where the SNP rate, s, is less than or equal to 0.4. For example, a sample of length 10 kb with $s = 0.01$ contains 100 SNPs, and so does a sample of 1 kb sequences with $s = 0.1$. A summary of data set characteristics is shown in Table 2-2.

Table 2-2. The characteristics of the simulated data sets. Each group is characterized by its sequence length (*l*), and the number of SNPs per site (*s*).

l **[kb]**	***S***	**Number of SNPs**
1	0.01, 0.1, 0.4	10, 100, 400
10	0.01, 0.1, 0.4	100, 1000, 4000
100	0.01, 0.1, 0.4	1000, 10000, 40000
1000	0.01, 0.1, 0.4	10000, 100000, 400000

For every configuration, 1000 sets of 10 sequences were generated. For each data set, all pairwise K_r values were correlated with the true substitution rate (the results obtained from the alignment-based distance-matrices). Finally, the correlation coefficient was computed for each configuration and averaged across all replicates.

The mean correlation coefficient between K_r and the alignment-based distance-matrix approaches 1 as the sequence length increases (Figure 2-5). For example, for the sample where l = 1 Mb, and s = 0.1, the mean correlation coefficient was 0.999951, and even the lowest mean correlation value (for the configuration where l = 1 kb, and s = 0.01) was still very high (0.922171).

Further, Figure 2-6 shows that the average correlation can be observed as a function of the total number of SNPs. Hence, the accuracy of K_r increases with the total number of SNPs in the data set, as long as the substitution rate is below 0.5.

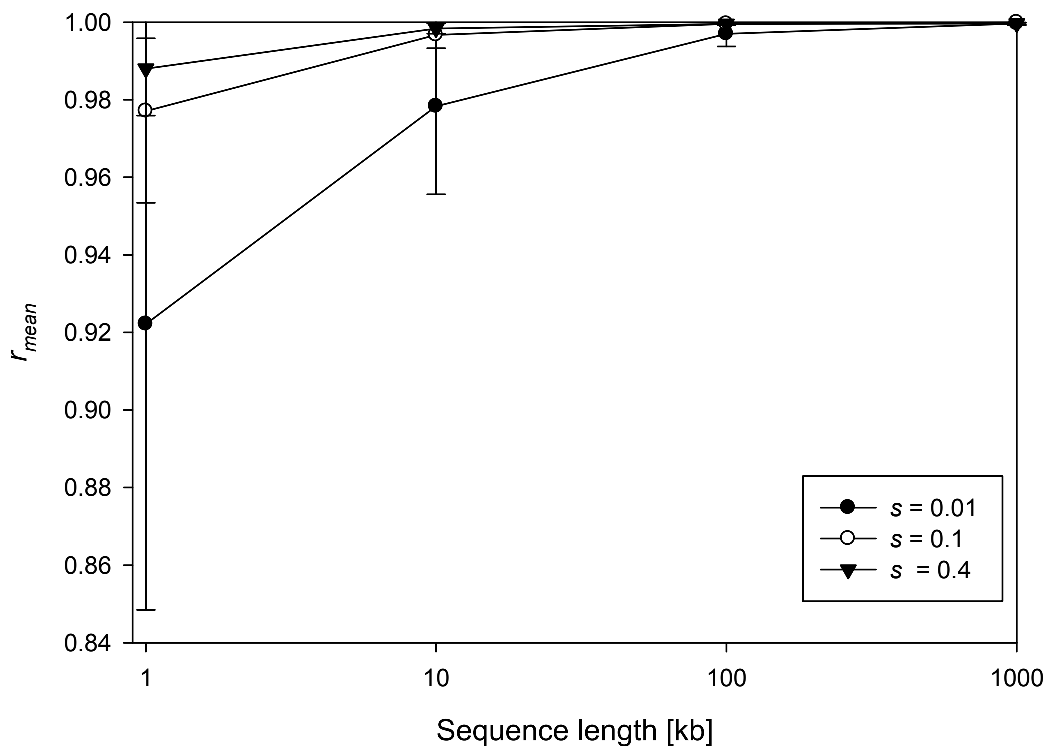


Figure 2-5. Correlation between K_r and the true substitution rate. The mean correlation coefficient (r_{mean}) ± SD is shown as the function of sequence length for different values of the SNP rate: s = 0.01, 0.1, 0.4. Each data point is computed from 1000 simulations.

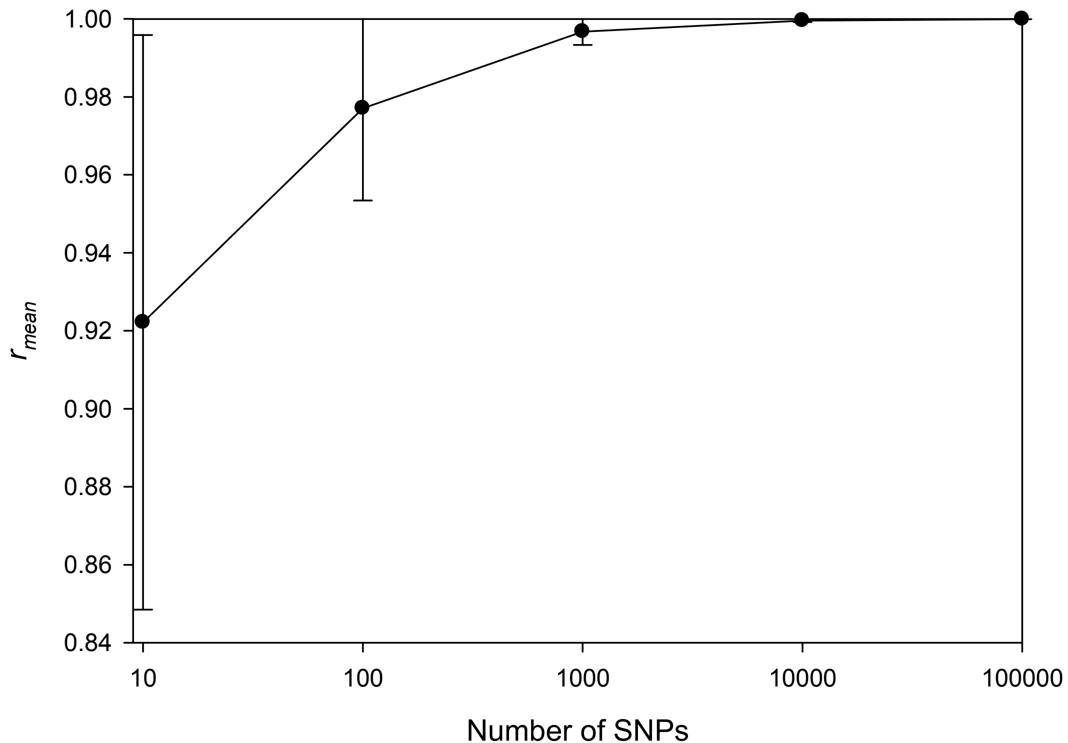


Figure 2-6. Correlation between K_r and the true substitution rate. The mean correlation coefficient (r_{mean}) ± SD is shown as the function of the number of SNPs contained in the sample. Each data point is computed from 1000 simulations.

2.3.3. The affect of horizontal gene transfer on the accuracy of K_r

Horizontal gene transfer (HGT) is a process in which genetic material from another organism is incorporated into the genome of the first organism, without the first organism being the offspring of the second one. Here, I discuss the affect of HGT on the accuracy of K_r.

Horizontal gene transfer was modeled for 100 simulated samples of 10 sequences of length 100 kb, and the number of SNPs per site s = 0.1. Each member of the sample received 1 to 10 chunks (substrings) from the random donor sequence. The length of each chunk was (i) 100 bp, and (ii) 1 kb. Each chunk was chosen from a random position of a random donor sequence. Figure 2-7 represents the results of the HGT simulation. The accuracy of K_r is not degraded by 100 bp chunks transfers, but

1 kb transfers significantly reduce the accuracy of K_r. However, the correlation mean is not below 0.91 even in the worst case scenario (10 transfers of 1 kb chunks).

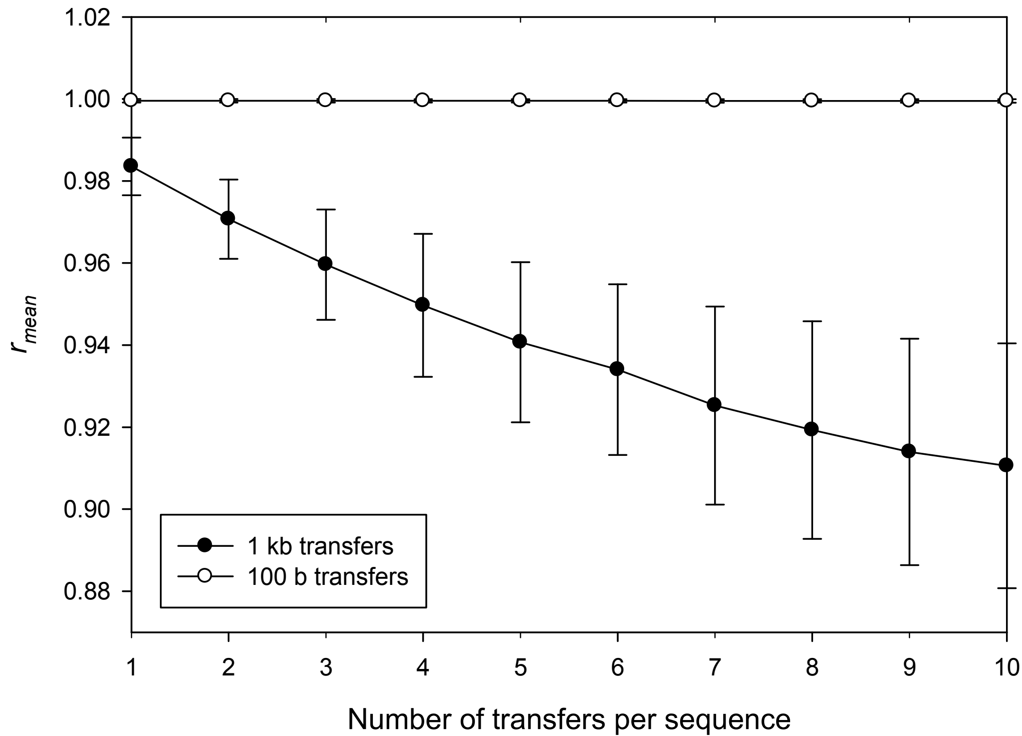


Figure 2-7. Horizontal gene transfer reduces the accuracy of K_r. The mean correlation coefficient (r_{mean}) ± SD is shown as a function of the number of transfers per sequence. The transfers of 100 bp chunks do not affect the accuracy of K_r, while the chunks of 1 kb do affect the accuracy of K_r. Each data point is computed from 100 simulations.

2.3.4. The effect of genome duplication on the accuracy of K_r

Gene duplication is a process by which a gene region is duplicated once or more times in the genome of an organism. The effect of gene duplication on the accuracy of K_r is explored in this section.

Gene duplication was modeled for 100 simulated samples of 10 sequences of length 100 kb, and the number of SNPs per site, $s = 0.1$. Each sequence from a data set was affected by 1–10 duplications of 1 kb chunks, where each chunk was copied

once, twice or five times. The first copy was inserted 1 kb downstream of the 3' end of the source. The subsequent copies were inserted 1 kb downstream of the 3' end of the previous copy. Figure 2-8 shows that the accuracy of K_r is only weakly degraded as the number of duplicated fragments, and/or the number of copies of a same fragment, increases.

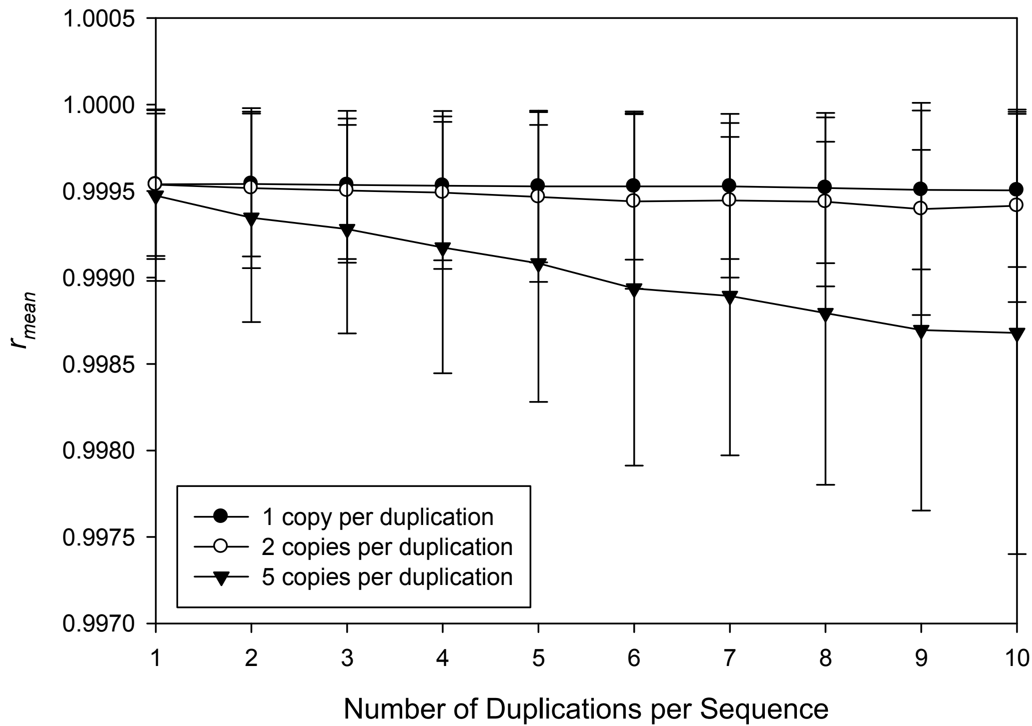


Figure 2-8. Gene duplication only weakly affects the accuracy of K_r. The mean correlation coefficient (r_{mean}) ± SD is shown as a function of the number of duplications per sequence. The transfers of 1 kb chunks which are copied 1, 2, or 5 times do not significantly affect the accuracy of K_r. Each data point is computed from 100 simulations.

2.3.5. Run time comparison of *kr* 1 and *kr* 2

The run-time behavior of the previous approach, implemented in *kr* 1, and the new approach, implemented in *kr* 2, is compared using simulated data sets. Each data set is characterized by its sequence length, l, the number of SNPs per site, s, and the

number of sequences in the set, n. The summary of data set characteristics is given in Table 2-3.

Table 2-3. Summary of group characteristics of the simulated data sets. Each group is characterized by its sequence length (l), the number of SNPs per site (s), and the number of sequences in the set (n).

l	***s***	***n***
10 kb	0.1	5, 10, 50, 100, 500, 1000
1 Mb	0.1	5, 10, 50, 100

Figure 2-9 shows the run-time comparison of the programs *kr* version 1 and *kr* version 2 for data sets with l = 10 kb. The regression lines are:

(i) for *kr* version 1: $R_1 = 0.0038 \times n^{1.9860}$

(ii) for *kr* version 2: $R_2 = 0.1070 \times n^{1.8931}$

Similarly, Figure 2-10 shows the run-time comparison of *kr* version 1 and *kr* version 2 for data sets with l = 1 Mb, and the corresponding regression lines are:

(i) for *kr* version 1: $R_1 = 1.6262 \times n^{2.2126}$

(ii) for *kr* version 2: $R_2 = 1.9924 \times n^{1.3508}$

Based on my previous time-complexity analysis (Sections 2.2.3 and 2.2.4), both *kr* 1 and *kr* 2 are expected to execute in $O(n^2l)$ time, and the program *kr* 2 is expected to execute faster because:

(i) the time required for the suffix tree construction step was reduced from $O(n^2l)$ time in the case of *kr* 1 to $O(nl)$ time in the case of *kr* 2

(ii) the time required for the computation of the observed average shustring lengths during a suffix tree traversal was reduced in the case of *kr* 2 due to the multiplication rule (see Algorithm 1, line 22)

The run-time behavior of both programs confirmed these expectations: *kr* 2 was faster than *kr* 1 for all simulated data sets. In the case of the data sets of length l = 10

kb, the regression lines for both *kr* 1 and *kr* 2 are close to $O(n^2 l)$, but with different constant factor. In this case, *kr* 2 is on average 25 times faster than *kr* 1.

More interestingly, the results for the data sets of 1 Mb sequences showed that the regression line of *kr* 2 stayed well below the upper bound of $O(n^2 l)$, and is $R_2 = 1.9924 \times n^{1.3508}$. For example, the comparison of run-times for 100 sequences of length 1 Mb (the total data set size is 100 Mb) resulted in 43-fold difference between *kr* 2 and *kr* 1: the execution of *kr* 2 took 17 minutes, and the execution of *kr* 1 took 722 minutes ≈ 12 hours (see Figure 2-10).

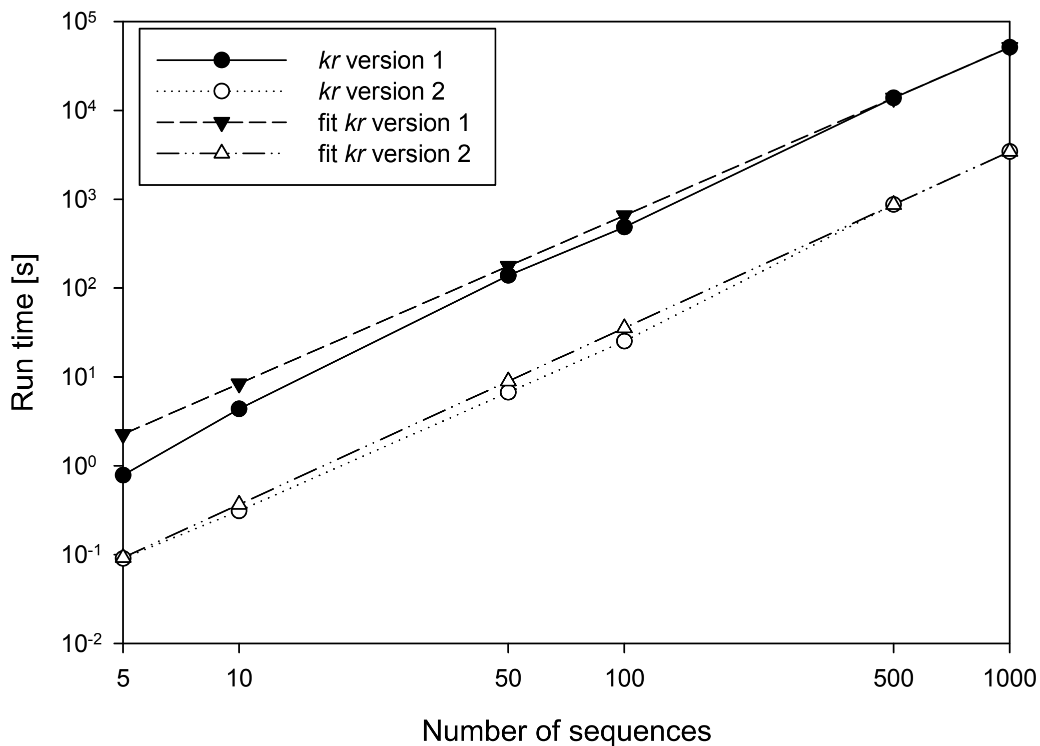


Figure 2-9. The run time comparison of *kr* version 1 and *kr* version 2 on 10 kb input sequences. The run time measured in seconds is shown as a function of the number of 10 kb input sequences.

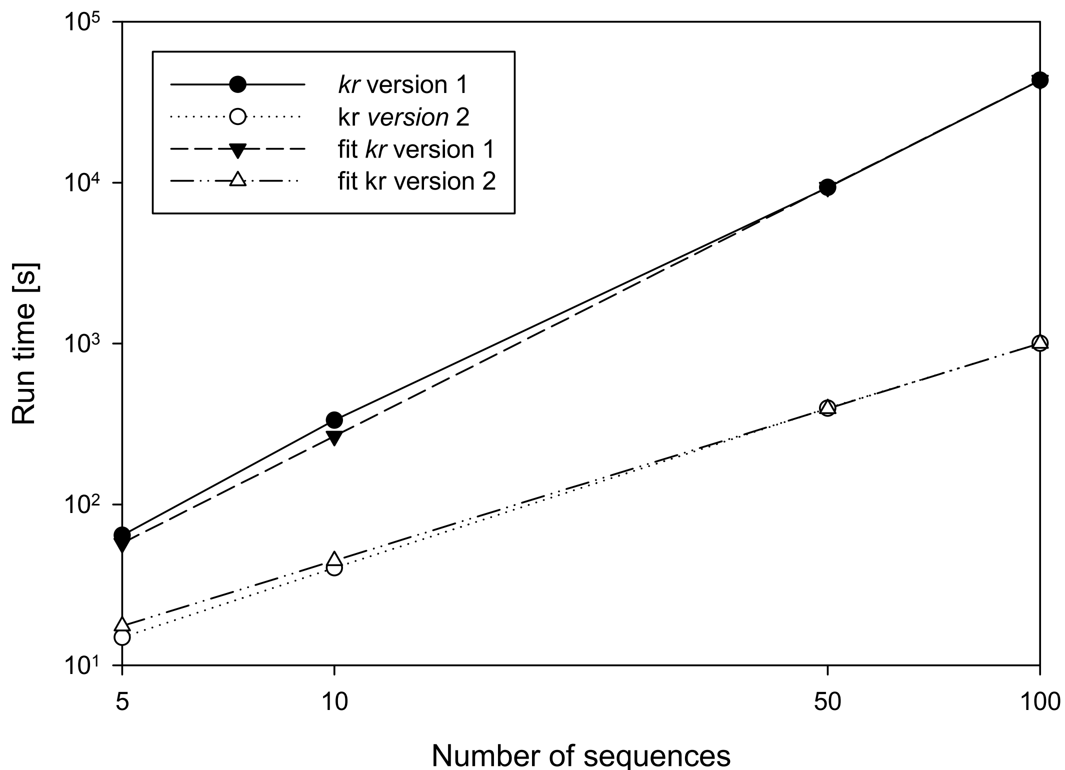


Figure 2-10. The run time comparison of *kr* version 1 and *kr* version 2 on 1 Mb input sequences. The run time measured in seconds is shown as a function of the number of 1 Mb input sequences.

2.4. Application of *kr* version 2

Finally, the accuracy and the scalability of *kr* 2 were compared to the best alignment-based results. Depending on the data set, K_r based distances were compared to the better of the results obtained by either MAVID (Bray and Pachter, 2004), or MUMmer (*Kurtz et al*. 2004). These programs are both very fast; MAVID is the MSA program suitable for the alignment of many short syntenic sequences, and MUMmer is the pair-wise alignment program designed for the comparison of large genomes (see Section 2.4.1). In addition, K_r-based distances were analyzed in comparison to the reference phylogenies derived from multiple sequence alignment of relevant genes, where reference phylogenies were available.

The following data sets of complete genomes were analyzed (Table 2-4):

(i) 12 *Drosophila* genomes (*Drosophila* 12 Genomes Consortium, 2007)

(ii) 13 *E*. *coli* and *Shigella* genomes (van Passel *et al*., 2008)

(iii) 825 HIV-1 pure subtype genomes (Wu *et al*., 2007)

Table 2-4. Analyzed data sets. The table contains the following statistics for each data set: the number of sequences in the data set (*n*), the average sequence length (l_{avg}), the size of the whole data set (*size*), the name of the alignment program that was used for the analysis of the data set, the time required for the execution of *kr* 2 (t_{kr2}), the *kr* 2 memory usage peak (m_{kr2}), the time required for the alignment computation (t_A), and the memory usage peak required for the alignment computation (m_A).

Data set	*n*	l_{avg} [Mb]	size [Mb]	alignment program	t_{kr2} [hh:mm:ss]	m_{kr2} [GB]	t_A [hh:mm:ss]	m_A [GB]
12 *Drosophila* genomes	12	169	2031	MUMmer	03:14:57	72	32:35:56	3.2
13 *E. coli* and *Shigella* genomes + 1 *Salmonella*	14	4.9	68	MUMmer	00:05:02	1.4	00:33:00	1.0
825 HIV-1 pure subtype genomes	825	0.009	7.5	MAVID	00:10:25	0.17	00:12:10	0.88

2.4.1. Auxililary software used for the analysis of real data sets

The analysis of 825 pure HIV-1 strains was compared to the results obtained using MAVID, which was the only MSA program scalable to a data set containing that many taxa. The other two data sets were analyzed using MUMmer, since it is a pair-wise sequence alignment software specifically developed for the comparison of large genomes, and was the only alignment tool able to efficiently compute the alignment of 12 *Drosophila* genomes (in total over 2 billion base pairs).

The memory consumption of both *kr* 2 and MUMmer is linear in the size of the data set, while the memory-consumption of MAVID grows worse than linearly. This makes MAVID inapplicable for the data sets of large genomes. Furthermore, MAVID does not cope well with the sequences which are not syntenic, or even properly assembled. In contrast, MUMmer deals well with sequence rearrangements, since it was designed for the computation of bacterial genomes, which frequently contain inversions.

Further, MUMmer generates only pair-wise alignments, and not phylogenetic distances between sequences. In order to compare K_r based distances to the results generated by MUMmer, I wrote an additional program (*parseDelta*) that estimates Jukes-Cantor distances from MUMmer alignments (Domazet-Lošo and Haubold, 2009).

The program *dnadist* from the PHYLIP package (Felsenstein, 1993) was used to compute the distance-matrices based on the Jukes-Cantor formula from the MSA produced by MAVID. The phylogenetic trees were constructed by applying the Neighbor-Joining method (Saitou and Nei, 1987), implemented in the program *neighbor* from the PHYLIP package (Felsenstein, 1993), and were drawn in MEGA 4 (Kumar *et al*., 2008).

The correlations between distance matrices based on K_r, and distance matrices based on alignments were assessed using Mantel's test implemented in the program *zt* (Bonnet and Van de Peer, 2002).

2.4.2. The analysis of 12 *Drosophila* genomes

The K_r based phylogenetic results for the large data set of 12 *Drosophila* genomes were compared to the generally accepted topology (*Drosophila* 12 Genomes Consortium, 2007), and to the pair-wise sequence alignment generated by MUMmer.

The analysis of 12 *Drosophila* genomes using *kr* 1 showed that the K_r distances produce the same topology as the reference topology (Haubold *et al*., 2009). Here, *kr* 2 confirmed these results (Figure 2-11), and showed the speed gain of *kr* 2 over *kr* 1. The program *kr* 2 took 3 hours and 15 minutes to compute K_r-based distances for 12 *Drosophila* genomes. In comparison, the run-time of *kr* 1 on the same test computer was approximately 2 days and 6 hours, which is roughly 16 times slower than *kr* 2.

However, the memory usage peak of *kr* 2 was 72 GB for this data set, compared to 13 GB required by *kr* 1. This drawback of *kr* 2 comes from the memory requirements of the 64-bit version of the program which was used for this data set. Namely, the total size of 12 *Drosophila* genomes is 2 Gb, which requires approximately 4 GB of memory space (each sequence is represented by its forward and reverse strand). The memory requirements of the underlying data structures of the 64-bit version are around 17-18 bytes per input character. For the data set of 12 *Drosophila* genomes this results in 72 GB memory usage peak (i.e. 4 GB $\times$ 18 = 72 GB).

Finally, the program MUMmer was also used for the analysis of this data set. The MUMmer based results gave a topology whose Symmetric Distance[5] (Robinson and Foulds, 1981) to the reference tree was 8. In addition, MUMmer took 6 times longer than *kr* 2 to compute the result ($\approx$ 1 day 8 hours).

[5] Symmetric Distance of Robinson and Foulds (1981) is a topological distance between a sequence pair (it does not take branch lengths into account). The distance equals the number of partitions which are present in one tree and not in the other.

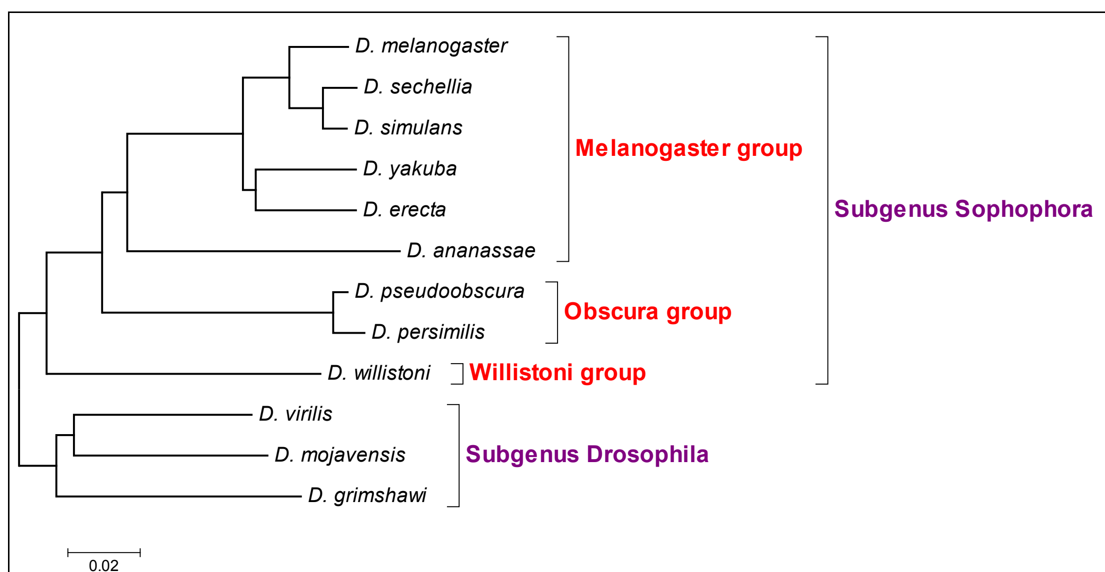


Figure 2-11. Phylogenetic tree of 12 *Drosophila* genomes based on K_r. The K_r-based evolutionary distances gave the same phylogenetic tree as the reference topology (*Drosophila* 12 Genomes Consortium, 2007).

2.4.3. The analysis of 13 *Escherichia coli* and *Shigella* genomes

The K_r based phylogeny of 13 enterobacterial genomes (13 *Escherichia coli* and *Shigella* genomes) was compared to both the MUMmer based results, and to the reference phylogeny (van Passel *et al.*, 2008). The size of genomes in the data set varies from 4.3 Mb to 5.5 Mb. As explained in Section 2.2.1, such differences in size affect the computation of observed average shustring lengths. This can be improved by excluding 95% of the shustrings observed by chance alone from the computation of average shustring lengths (see Section 2.2.1, Figure 2-1; Haubold *et al.*, 2009).

Figure 2-12 shows the neighbor-joining trees of complete genomes based on K_r distances (Figure 2-12c), MUMmer-derived distances (Figure 2-12b), and the reference phylogenetic tree (Figure 2-12a), based on the multiple sequence alignment of 169 single-copy genes (van Passel *et al.*, 2008) computed using MAFFT (Katoh and Hiroyuki, 2008).

The computation of K_r-based distances of 13 enterobacterial genomes with *kr* 2 finished in 5 minutes 2 seconds, which is over 10 times faster than the analysis of the data set with *kr* 1, which took 59 minutes. However, the K_r-based phylogenetic tree differs from the referenced phylogeny in the position of *E. coli* UTI89, marked with * (Figure 2-12c). The strain *E. coli* UTI89 clusters with *E. coli* 536 in the case of K_r, and with *E. coli* CFT073 in the case of the reference phylogeny. In comparison, MUMmer-based distances yielded the correct phylogeny, with the execution time of 33 minutes, which was over 6 times longer than *kr* 2. The memory consumption peak of *kr* 2 was 1.4 GB, and 1 GB in the case of MUMmer.

Finally, the K_r-based and the MUMmer-based distances were computed from the concatenation of 169 selected genes used by van Passel *et al*. (2008). The neighbor-joining trees based on both methods produced the same topology as the referenced one (van Passel *et al*., 2008).

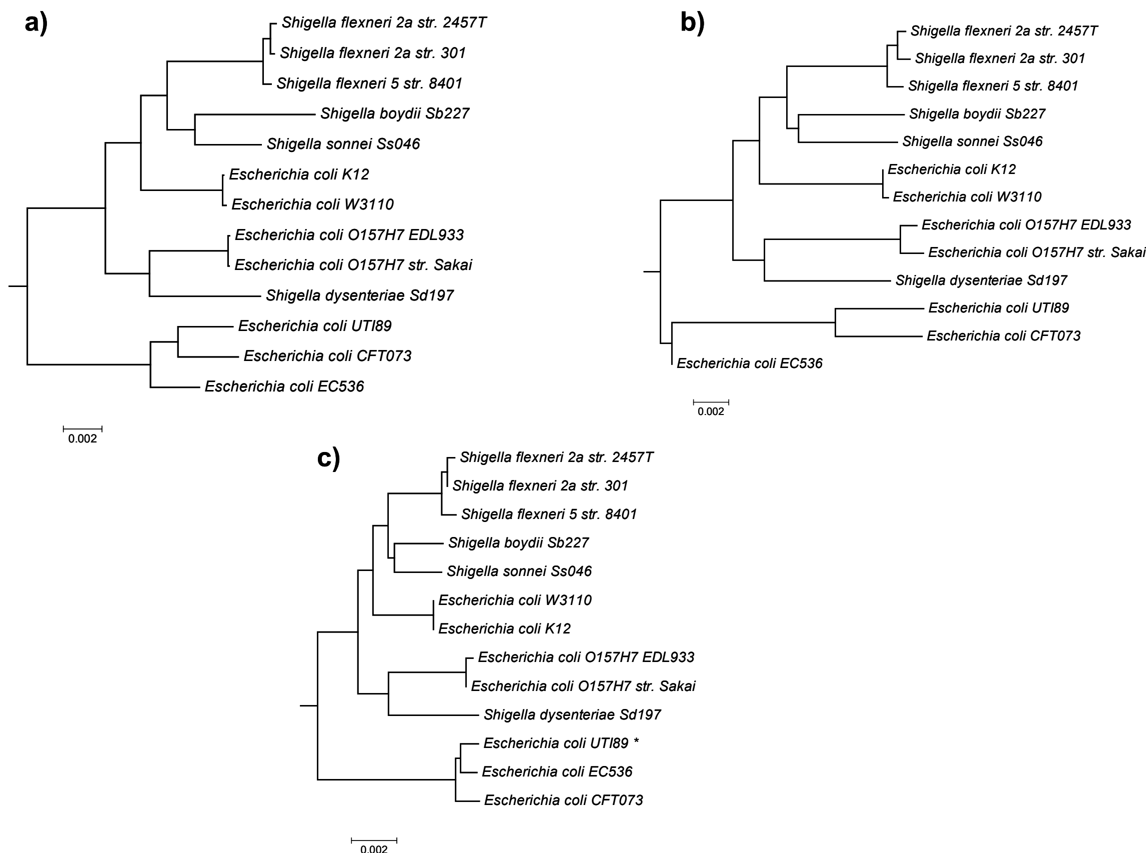


Figure 2-12. Phylogenetic trees of 13 strains of *Escherichia coli* and *Shigella*. a) Reference phylogeny based on the multiple sequence alignment of 169 genes (van Passel *et al*., 2008). **b)** Whole-genome phylogeny based on the pair-wise alignment computed by MUMmer (Kurtz *et al*., 2004). **c)** Whole-genome phylogeny based on K_r; the strain *E.coli* UTI89, marked with *, is clustered with *E.coli* 536, while in the reference phylogeny it clusters with *E.coli* CFT073.

2.4.4. The analysis of 825 HIV-1 pure subtype genomes

In the last example, the genomes of 825 pure HIV strains were analyzed (Wu *et al.*, 2007). The K_r-based distances between 825 HIV-1 strains were compared to the results obtained using the MSA program MAVID (Bray and Pachter, 2004), which was the best tool available for the comparison of many short, syntenic[6] sequences. The resulting phylogenies are displayed in Figure 2-13. There is an excellent correlation (r = 0.97; p = 0.001) between the K_r-based and the MAVID-based distance-matrices.

However, in the case of the K_r-based phylogeny, strain A_DQ083238, officially classified as an A strain, was clustered with the C strains (see Figure 2-13). On closer inspection, this strain was discovered to be a recombinant with most of the genome derived from an A1 strain and a C strain. This example indicated the limitation of K_r as a global similarity measure: Here, in the case of A_DQ083238, the strong similarity between a C strain and a part of A_DQ083238 prevailed over the similarity between an A1 strain and the rest of the genome.

The computation of K_r-based distances with *kr* 2 took 10 minutes 25 seconds, which was almost 20 times faster than the execution time of *kr* 1 (3 hours 26 minutes). In comparison, MAVID classified all strains correctly and swiftly in 12 minutes 10 seconds.

[6] Syntenic literally means on the same band (ribbon). In this context, synteny refers to the preserved order of genes along chromosomes (in the comparison of related species).

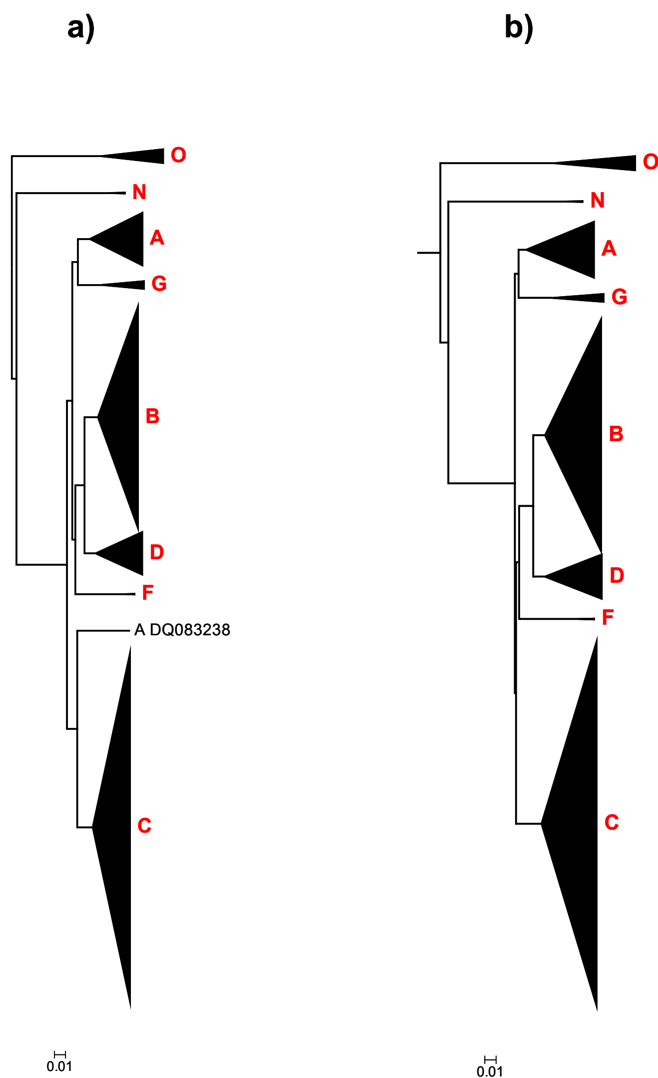


Figure 2-13. Phylogenetic trees of 825 "pure" HIV-1 strains. The strains of the same subtype are represented within the compressed subtree of the subtype. **a)** Whole-genome phylogeny based on K_r. Only a single strain, A_DQ083238, was not classified according to its official subtype (subtype A). It is actually a recombinant between an A and a C strain. **b)** Whole-genome phylogeny based on the multiple sequence alignment computed by MAVID. All strains were classified according to their official subtype.

2.5. Discussion

Evolutionary relationships between organisms can be estimated by aligning sequences and then deriving evolutionary distances from the alignment. These

distances are then summarized as phylogenetic trees. As an alternative to alignment, alignment-free methods can be used. Here, I stress two possible applications of alignment-free methods: (i) the fast computation of guide trees for progressive alignment methods, and (ii) the analysis of sequences with large scale rearrangements (e.g. sequences with low synteny or unassembled genomes).

However, alignment-free methods generally do not yield evolutionary distances (i.e. substitution rates). To fill this gap, we have recently developed an alignment-free estimator of substitution rates between sequence pairs, K_r, which outperformed other state-of-the art alignment-free methods for closely related sequences (Haubold *et al.*, 2009). The analysis of simulated data showed that K_r-based evolutionary distances are accurate for closely related sequences, when the substitution rates are less than 0.5 (Haubold *et al.*, 2009). Here, I showed that the accuracy of K_r-based evolutionary distances increased with the total number of single nucleotide polymorphisms (SNPs) in the data set (Section 2.3.2; Figures 2-5 and 2-6), where the number of SNPs is determined by both the length of sequences in the data set, and the substitution rate between sequences (Domazet-Lošo and Haubold, 2009). In contrast, horizontal gene transfer degraded the accuracy of K_r-based evolutionary distances (Section 2.3.3). This result corresponds to the example of a misclassified recombinant HIV-strain: a strain was clustered with C strains, although only 30% of its genome derives from a C strain, with rest mostly of an A subtype (Section 2.4.4; Figure 2-13).

The K_r-based distances are computed from the observed average shustring lengths between a sequence pair. In the previous implementation of the method, *kr* version 1, every pairwise distance was computed by constructing and traversing a generalized suffix tree of a pair of sequences (Haubold *et al.*, 2009). This resulted in the construction of $n(n\text{-}1)$ suffix trees for n sequences of length l. The time required for the construction of all suffix trees was $O(n^2l)$. Similarly, the time required for the traversal of all suffix trees with the computation of all K_r-based distances was $O(n^2l)$. This was too slow for large data sets.

To address this problem, in the first part of my thesis I developed an algorithm that computes all $\binom{n}{2}$ pair-wise K_r distances in a single traversal of a generalized

suffix tree of *n* sequences (Algorithm 1, Section 2.2.5), and implemented in the program *kr* version 2 (Domazet-Lošo and Haubold, 2009). In this way, the time complexity of the suffix tree construction step was reduced from $O(n^2 l)$ to $O(nl)$. The step of traversing a suffix tree with computing all distances remained $O(n^2 l)$ in both the new and the old approach, but the number of operations in the new approach is upper-bounded by the number of operations of the old approach (see multiplication rule of Algorithm 1, line 22). In comparisons of these two approaches using simulated data sets, *kr* 2 was in all cases at least ten times faster than *kr* 1 (Section 2.3.5). For example, *kr* 2 was more than 40 times faster than *kr* 1 for the data set of 100 sequences of length 1 Mb (the total size of the data set was 100 Mb), i.e. the execution time of *kr* 2 was 17 minutes, while the execution time of *kr* 1 was 12 hours (Section 2.3.5; Figure 2-10). In addition, *kr* 2 was at least 10 times faster than *kr* 1 in the analysis of real data sets: the computation of 12 *Drosophila* genomes was sped up 16 times, the analysis of 825 HIV-1 genomes 20-fold, and the analysis of 13 enterobacterial genomes was sped up 10-fold (Section 2.4).

To further assess the accuracy of the K_r metric, and to test the scalability of its implementation in *kr* 2, I compared *kr* 2 to two popular alignment tools, MUMmer (Kurtz *et al*., 2004), and MAVID (Bray and Pachter, 2004) (Section 2.4). MUMmer is an efficient pair-wise alignment tool designed for the comparison of large genomes, while MAVID is a multiple sequence alignment tool scalable to the data sets of numerous short syntenic sequences. I showed that the program *kr* 2 is scalable to both of these extremes. First, *kr* 2 was compared to both MUMmer and the available reference phylogenies on the data set of 12 *Drosophila* unassembled genomes (with the total size of over 2 billion base pairs) (Section 2.4.2), and on the data sets of 13 enterobacterial genomes (Section 2.4.3). The genomes in both data sets are affected by horizontal gene transfer, recent duplication and large-scale rearrangements. Second, I compared *kr* 2 to MAVID on the data set of 825 HIV-1 genomes (Section 2.4.4).

The K_r-based evolutionary distances yielded the best results in the case of the 12 *Drosophila* genomes: the K_r-based phylogeny was topologically identical to the reference phylogeny (*Drosophila* 12 Genomes Consortium, 2007), which was not the case with the MUMmer phylogeny. This agrees with the previous studies where

some alignment-free methods outperformed alignment-based results on rearranged input sequences (Höhl *et al*., 2006; Sims *et al*., 2009). However, for the data sets of 13 enterobacterial genomes, and the data set of 825 HIV-1 genomes, the alignment based methods gave better results than K_r. In the case of the K_r-based phylogeny 825 HIV-1 genomes, a single strain was misclassified (Section 2.4.4; Figure 2-13). Nevertheless, the correlation between the K_r-based and the MAVID-based distances was very high ($r = 0.97$, $p = 0.001$).

Further analysis of the HIV-1 strain misclassified by K_r revealed its recombinant nature. Motivated by this result, I investigated the application of shortest unique substrings in the context of local sequence homology (Chapter 3).

3. Efficient Alignment-Free Detection of Local Sequence Homology

3.1. Introduction

In biology, similar sequences usually have similar (or the same) functions due to their common evolutionary history (Chapter 1). In order to determine sequence similarity two approaches can be used: (i) sequences can be compared using global measures of similarity (see Chapter 2), and (ii) sequences can be compared in order to find locally similar regions, which can be further used for inferring local sequence homology.

Local sequence similarity is typically determined by alignment methods: an optimal pair-wise alignment can be computed using the Smith-Waterman algorithm (Smith and Waterman, 1981; Chapter 1) or more efficient heuristic methods may be used. Two popular heuristic local alignment methods are BLAST (Altschul *et al.*, 1990)[7] and FASTA (Lipman and Pearson, 1985). They compare subsequences (*words*) of a query sequence to a large database of sequences. When a good match between a subsequence of a query and a database entry is found, it is extended to the left and to the right in order to get a longer match, and several significant matches could be combined to form a single one. Both programs return scored alignments between a query and the database sequences.

Local sequence similarity is particularly important for detecting conserved regions (e.g. genes) among otherwise dissimilar sequences. In particular, closely-related organisms may have many similar regions, or they can even be similar across their complete genomes (the problem of global sequence similarity; see Chapter 2), while distantly-related organisms may have only isolated regions of similarity. Thus, the detection of local sequence similarity is of special relevance for comparison of distantly-related sequences. However, solving this problem is also important for detecting genetic recombination, which usually involves more closely-related sequences.

[7] The importance of these tools can be best illustrated by the number of citations: the original BLAST publication (Altschul *et al.*, 1990) was the most cited publication in the 1990s, with more than 28,064 citations by June 12, 2010 (ISI Web of Knowledge).

Genetic recombination is a process where the genetic material of different organisms is combined together. Homologous recombination is the exchange of homologous (related) sequence parts, as opposed to non-homologous recombination, where the evolutionary non-related parts are exchanged. Further, recombination can be either reciprocal or non-reciprocal (Figure 3-1). In the first case, sequences reciprocally exchange genetic material (Figure 3-1a). In the second case, genetic information is transferred only in one direction: from a donor sequence to a recipient sequence (Figure 3-1b).

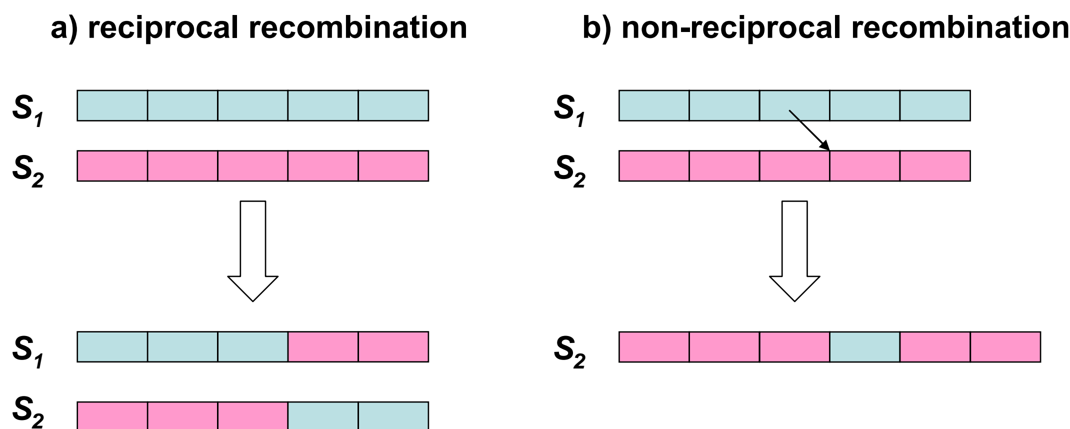


Figure 3-1. Reciprocal and non-reciprocal recombination. In the case of reciprocal recombination (DNA crossover), homologous genetic material is exchanged between sequences S_1 and S_2 (shown as the exchange of 2 blue and 2 pink rectangles). In the case of non-reciprocal recombination, a part of a donor sequence, S_1, is incorporated in a recipient's sequence, S_2 (shown as the blue rectangle between pink sequences in the final form of S_2).

In eukaryotes[8], recombination occurs during meiosis between parental DNA sequences to produce gametes. This homologous reciprocal recombination is also known as DNA crossover (Figure 3-1a). Another form of homologous reciprocal recombination occurs when a host cell is coinfected by more than one viral strain, and a recombinant form of the virus is generated. An example for this is coinfection of a human cell with different strains of the human immunodeficiency virus, HIV. A

[8] A eukaryote is an organism whose cell contains nucleus inside its membranes (animals, plants, and fungi), as opposed to prokaryotes which do not have a cell nucleus (e.g. bacteria).

new recombinant form of HIV, when detected in a certain number of unrelated cases, is then referred to as a circulating recombinant form (CRF)[9].

In contrast, gene conversion and horizontal gene transfer (HGT) (or lateral gene transfer; LGT) are types of non-reciprocal recombination (Figure 3-1b). In the case of gene conversion, genetic material is transferred within an organism, while in the case of HGT, genetic material from a donor organism is incorporated into a recipient's genome. HGT has been more frequently observed in bacterial genomes, but it is also known for eukaryotic genomes (Keeling and Palmer, 2008).

In general, recombination implies phylogenetic incongruencies along a genome, that is, the phylogeny of some regions (e.g. genes) can disagree with the phylogeny of the whole organisms (Dykhuizen and Green 1991; Posada *et al*. 2002; Keeling and Palmer, 2008). Detecting phylogenetic incongruence is often challenging. Moreover, Posada and Crandall (2001) showed that the performance of the methods for recombination detection can vary under different conditions (e.g. genetic divergence; the number of recombinational events). Thus, the choice of an appropriate method should depend on the data set analyzed (Posada and Crandall, 2001).

Besides detecting recombination event, some methods can also detect recombinational breakpoints and the sequences involved in the recombinational event: parental and recombinant sequences (e.g. Boni *et al*., 2007). In particular, a subgroup of the recombination detection methods was developed for determining the subtypes or a mosaic form of a query sequence (e.g. Rozanov *et al*. 2004; Wu *et al*. 2007; Kosakovsky Pond *et al*., 2009). These methods are called subtyping methods, and are typically used for the classification of new viral (e.g. HIV) sequences. Among the subtyping tools developed for HIV classification, the phylogeny-based subtyping methods provide more accurate results (e.g. Kosakovsky Pond *et al*., 2009), but are slow when applied to large data sets. As an alternative, less precise, but more efficient alignment-free approaches can be used, e.g. the subtyping based on BLAST scores (Rozanov *et al*. 2004) or the approach based on the distribution of short nucleotide sequences among different HIV subtypes (Wu *et al*. 2007), etc.

[9] The list of HIV circulating recombinant forms (CRFs) is available at http://www.hiv.lanl.gov/content/sequence/HIV/CRFs/CRFs.html.

As explained in Section 2.2.1, we have recently developed an alignment-free estimator of substitution rates between sequence pairs, based on the average length of shortest unique substrings between sequences, K_r (Haubold *et al*. 2009). In K_r-based clustering of 825 HIV-1 genomes, all strains, except one, were correctly classified as their official subtype (Section 2.4.4; Figure 2-13). In further analyses, we discovered that the recombinant form of the misclassified strain was a recombinant (Domazet-Lošo and Haubold, 2009; Sections 2.4.4 and 3.4.2). Taking this result as a starting point, in this part of my thesis, I address the problem of detecting locally similar regions between sequences. As one part of the solution to the problem, I developed a new algorithm for the detection of matching regions between sequences (Section 3.2.4) based on shortest unique substrings (which were described in Section 2.2.1). I implemented this algorithm in the subtyping program *st*, which was then used for recombination detection in both simulated and real data sets. To illustrate the efficiency and the scalability of the approach, the program was used for the recombination detection in the pathogenic bacterium *Neisseria meningitidis*, to classify circulating recombinant forms of HIV, and to find the closest relative of an avian pathogenic *Escherichia coli* strain between 13 *Escherichia coli* and *Shigella* strains (Section 3.4).

3.2. Methods

3.2.1. Problem statement – determining subtype(s) of a query sequence

Let $\boldsymbol{Q} = \{Q_1, .., Q_m\}$ be a set of m, and $\boldsymbol{S} = \{S_1, .., S_n\}$ be a set of n (closely related) nucleotide sequences represented as strings over the alphabet {A, C, G, T}, and terminated by a unique character as before. Again, each sequence from $\boldsymbol{Q}$ and $\boldsymbol{S}$ is represented by its forward and reverse strand. Each sequence from $\boldsymbol{Q}$ is considered as *query*, and each sequence from $\boldsymbol{S}$ is considered as *subject*. Again, without loss of generality, let l denote the length of each sequence.

Let each query sequence Q_i be represented by a list G_i, where G_i consists of k ($k \geq 1$) segments: $G_i = G_{i,1} .. G_{i,k}$, where:

(i) $G_{i,1}$ corresponds to the substring $Q_i[1 .. p_2 - 1]$

(ii) $G_{i,d}$ corresponds to the substring $Q_i[p_d .. p_{d+1} - 1]$ for $d = 2, .., k -1$

(iii) $G_{i,k}$ corresponds to the substring $Q_i[p_k .. |Q_i|]$

and every segment $G_{i,d}$ of G_i is annotated by $\boldsymbol{S_{i,d}}$, $\boldsymbol{S_{i,d}} \subseteq \boldsymbol{S}$, such that (see Figure 3-2):

(i) among members of $\boldsymbol{S}$, members of $\boldsymbol{S_{i,d}}$ are the most similar to Q_i over the segment $G_{i,d}$

(ii) $\boldsymbol{S_{i,d}} \neq \boldsymbol{S_{i,d-1}}$, for $d = 2, .., k$

The task is to find G_i as a list of segments $G_{i,d}$, $d = 1, .., k$. Notice that the number of segments k is not known in advance, and should be automatically detected.

	1	2	3	4	5
S_1	G	A	$		
S_2	G	C	C	$	
S_3	T	A	$		
Q_1	T	A	G	C	$

$G_{1,1}$ $G_{1,2}$

Figure 3-2. Q_1 represented by a list $G_1 = G_{1,1}\ G_{1,2}$. Q_1 is a query and $S = \{S_1, S_2, S_3\}$ is the set of subject sequences. In the context of local sequence similarity, Q_1 can be represented by two segments $G_{1,1}$ and $G_{1,2}$. The first segment $G_{1,1}$ corresponds to the substring $Q_1[1..2]$, which is the most similar to the substring of S_3, $S_3[1..2]$. The second segment $G_{1,2}$ corresponds to the substring $Q_2[3..4]$, which is the most similar to the substring of S_2, $S_2[1..2]$. Hence, Q_1 can be observed as a mosaic of S_3 and S_2.

A segment $G_{i,d}$, $d = 1, .., k$, that corresponds to a substring $Q_i[lb .. rb]$, is described by the following items:

(i) lb – the left endpoint of the segment

(ii) rb – the right endpoint of the segment

(iii) $\boldsymbol{S_{i,d}}$ – the subset of $\boldsymbol{S}$, whose members are the most similar to the substring $Q_i[lb\,..\,rb]$; that is, Q_i is considered to be locally homologous to members of $\boldsymbol{S_{i,d}}$ over $G_{i,d}$

Further, if a query Q_i is represented by a single segment $G_{i,1}$ that is most similar to the members of a subset $\boldsymbol{S_{i,1}}$, then we say that Q_i is of *subtype* $\boldsymbol{S_{i,1}}$. If a query Q_i is represented by k segments, $k > 1$, then we consider Q_i to be a *recombinant* (or a *mosaic*) of different subtypes, $\bigcup_{d=1}^{k} \boldsymbol{S_{i,d}}$.

3.2.2. Construction of locally homologous segments

Let Q_i and S_j be a pair of sequences from $\boldsymbol{Q}$ and $\boldsymbol{S}$, respectively. Let $h_{i,j,p}$ denote the shortest unique substring (shustring), as before (see Section 2.2.1). Let $H_{i,p}$ denote the greatest value of $|h_{i,j,p}|$ at position p, that is, $H_{i,p} = \max\{|h_{i,j,p}|\}$. Let $\boldsymbol{S_{i,p}}$ denote the subset of $\boldsymbol{S}$ such that:

$$\boldsymbol{S_{i,p}} = \{S_j \in \boldsymbol{S} \mid |h_{i,j,p}| = H_{i,p}\}.$$

Values $H_{i,p}$ and $\boldsymbol{S_{i,p}}$ need to be stored for each position p, $p = 1, \ldots, |Q_i|$. For efficient memory usage, the following approach is used: an interval $I_{i,p}$ is formed at position p, and contains the following fields:

(i) lb – the left endpoint of the interval: $lb = p$

(ii) rb – the right endpoint of the interval; initially: $rb = lb + H_{i,p} - 1$

(iii) sl – shulen at the beginning of the interval: $sl = H_{i,p}$

(iv) $\boldsymbol{S_{sl}}$ – a subset of $\boldsymbol{S}$, such that $\boldsymbol{S_{sl}} = \{S_j \in \boldsymbol{S} \mid |h_{i,j,p}| = H_{i,p}\}$

Interval $I_{i,p}$ is denoted by $[I_{i,p}.lb, I_{i,p}.rb]$. The important property of the final form of an interval $I_{i,p}$ is that the value $H_{i,p'}$ for every p', $p' = 1, .. ,|Q_i|$, can be reconstructed from $H_{i,p}$, as:

$$H_{i,p'} = H_{i,p} - (p' - I_{i,p}.lb) \text{ for } I_{i,p}.lb \leq p' \leq I_{i,p}.rb.$$

Notice that when an interval $I_{i,p}$ is formed, the value at position p', $I_{i,p}.lb \leq p' \leq I_{i,p}.rb$, determined as $H_{i,p} - (p' - I_p.lb)$, does not have to be equal to $H_{i,p'}$. In that case, when a new interval $I_{i,r}$ is formed starting at position r, where $I_{i,p}.lb \leq r \leq p' \leq I_{i,p}.rb$:

(i) the value at position p' is readjusted to $H_{i,r} - (p' - I_{i,r}.lb)$

(ii) the right endpoint of interval $I_{i,p}$ is readjusted: $I_{i,p}.rb = I_{i,r}.lb - 1$

Algorithm 2, which constructs an interval list I_i for every sequence Q_i and a set $\boldsymbol{S}$, is presented in Section 3.2.4. An interval list I_i consists of t elements (intervals), $I_i = I_{i,1}$, .., $I_{i,t}$, $t \leq |Q_i| = l$. Since t is at least several times smaller than l both for simulated and real data sets, this provides overall memory reduction over an approach where $H_{i,p}$ and $\boldsymbol{S}_{i,p}$ are stored for every position p.

Finally, I_i is transformed into a list of segments $G_i = G_{i,1} ... G_{i,k}$. For every segment $G_{i,d}$ ($d = 1, .., k$), a set of subjects $\boldsymbol{S}_{i,d}$ is determined as a set of subjects that are most locally similar to Q_i over a segment $G_{i,d}$. The construction of G_i from I_i using a sliding window analysis is described in Section 3.2.5.

3.2.3. Time complexity of computing a list of intervals I_i

In order to compute the list of intervals I_i for the query sequence Q_i, the values of $H_{i,p}$ have to be found for every position $p = 1, .., |Q_i|$. To determine $H_{i,p}$, a shustring $h_{i,j,p}$ needs to be found for every S_j.

To find all $h_{i,j,p}$, $p = 1, .. ,|Q_i|$, a generalized suffix tree T for a pair of sequences (Q_i, S_j) is constructed and traversed. In T, a terminal node (Q_i, p) corresponds to a suffix $Q_i[p .. |Q_i|]$, and a terminal node (S_j, r) corresponds to $S_j[r .. |S_j|]$. A shustring $h_{i,j,p}$ is the shortest prefix of $Q_i[p .. |Q_i|]$ absent from $S_j[r .. |S_j|]$, and is found in T as the path label of a branch node v, that is the lowest common ancestor of (Q_i, p) and (S_j, r), plus the first character on the path from v to (Q_i, p).

The time needed to construct and traverse a suffix tree is proportional to the total length of the input text. When a generalized suffix tree is constructed of a single query sequence (Q_i) and n subject sequences (from $\boldsymbol{S}$), then in a single tree traversal all shustrings $h_{i,j,p}$ for every $S_j \in \boldsymbol{S}$, and for every position p = 1, .., $|Q_i|$ can be determined. The time-complexity of the construction and the traversal of a generalized suffix tree of n subject sequences and a query sequence is $O(nl)$. If the computation is extended to m query sequences, then the construction and the traversal of the corresponding generalized suffix tree is $O((m + n)l)$.

I describe a new algorithm, Algorithm 2, that constructs an interval lists for every query Q_i during a single traversal of the generalized suffix tree T of all queries and subjects. More precisely, instead of an interval list, an interval tree, IT_i, is constructed. An interval tree is a binary tree of intervals. An interval-list can be obtained from an interval tree by traversing the tree inorder.

In Algorithm 2, an interval tree is used instead of an interval list, since inserting an interval in an interval-list takes $O(l)$ time, and inserting an interval-node in an interval-tree takes $O(\log l)$ time. Thus, the overall time to construct an interval tree for a single query takes $O(l \log l)$.

The construction of m interval trees for m queries during a single traversal of a generalized suffix tree of m queries and n subjects takes $O((m + n)l + ml \log l) = O(l (n + m(1 + \log l)))$ time.

3.2.4. Algorithm 2: Construction of an interval tree

An interval tree IT_i for Q_i consists of non-overlapping interval-nodes. An interval-node $z_{i,p}$ of IT_i corresponds to an interval $I_{i,p} = [I_{i,p}.lb, I_{i,p}.rb]$ from I_i, and contains following elements:

(i) lb – the left endpoint of the interval-node; $z_{i,p}.lb = I_{i,p}.lb$

(ii) rb – the right endpoint of the interval; $z_{i,p}.rb = I_{i,p}.rb$, and initially: $z_p.rb = z_p.lb + H_{i,p} - 1$

(iii) sl – shulen at the beginning of the interval: $sl = H_{i,p}$

(iv) $\boldsymbol{S_{sl}}$ – a subset of $\boldsymbol{S}$, such that $\boldsymbol{S_{sl}} = \{S_j \in \boldsymbol{S} \mid |h_{i,j,p}| = H_{i,p}\}$

(v) *left* – pointer to the left child of $z_{i,p}$

(vi) *right* – pointer to the right child of $z_{i,p}$

Each interval-node $z_{i,p}$ is formed in such a way that the following rules hold:

(i) for every interval-node x in the left subtree of $z_{i,p}$: $x.rb < z_{i,p}.lb$

(ii) for every interval-node x in the right subtree of $z_{i,p}$: $z_{i,p}.rb < x.lb$

An interval node $z_{i,p}$ is denoted by $[z_{i,p}.lb, z_{i,p}.rb]$.

The construction of an interval-tree IT_i for Q_i is described in Algorithm 2 (Figure 3-3). IT_i is constructed while traversing a generalized suffix tree T of Q_i and n subject sequences, $\boldsymbol{S} = \{S_1, .., S_n\}$.

Notice, at the moment of insertion of an interval-node in an interval-tree, an interval can have unadjusted endpoints, that is, a new interval-node can overlap with one or more intervals-nodes in IT_i. In that case, the endpoints of both a new interval and the intervals in IT_i are adjusted, so that the tree always contains non-overlapping intervals.

Let every **branch node $\boldsymbol{v}$** of T contain following fields:

(i) *subjectId* - the set of subject identifiers $\{S_j \mid \exists\ (S_j, p)$ that is a terminal node in the subtree rooted on $v\}$; notice that this list is empty for the branch nodes that contain only terminal nodes that refer to the query

(ii) *branchChildren*: the set of branch nodes which are children of v

(iii) *termChildren*: the set of terminal nodes which are children of v

(iv) *stringDepth*: the length of the path label of v; the string depth of v represents the length of the longest common prefix of all suffixes corresponding to the terminal nodes in the subtree rooted on v

(v) *unresolvedTerm*: the set of unresolved terminal nodes of v that refer to the query

I call a terminal node that refers to a query a *query terminal node*, and a terminal node that refers to a subject a *subject terminal node*. A query terminal node (Q_i, p) is

said to be resolved, when the value $H_{i,p}$ is determined for this node. The set *unresolvedTerm* contains query terminal nodes in the subtree rooted on *v* for which $H_{i,p}$ has not yet been determined. That is, there are no subject terminal nodes in the subtree rooted on *v*, so the value $H_{i,p}$ could not have been determined for any terminal node *w* in the subtree rooted on *v* ($w \in v.unresolvedTerm$).

Let every **terminal node *w*** contain the following fields:

(i) *seqId*: the sequence identifier referred to by *w* (Q_i or S_1, .., S_n)

(ii) *pos*: starting position of the suffix (*seqId*[*pos* .. |*seqId*|]) referred to by *w*

```
Algorithm 2 Construct an interval tree
Require: T {suffix tree of n + 1 DNA sequences Q1,S1,S2,...,Sn}
Ensure: IT {interval tree}
 1:  traverse(root(T), root(IT))
 2:  updateITree(root(IT), -1)
 3:  function traverse(v, z)
 4:  for all w ∈ v.branchChildren do
 5:    traverse(w, z)
 6:  if v.subjectId is not empty then
 7:    for all w ∈ v.termChildren do
 8:      if w is a query node then
 9:        formIntervalNode(w, v, z)
10:    for all w ∈ v.unresolvedTerm do
11:      formIntervalNode(w, v, z)
12: end function

13: function formIntervalNode(w, v, z)
14:   new.lb = w.pos
15:   new.sl = v.stringDepth + 1
16:   new.Ssl = v.subjectId
17:   addIntervalNode(z, new)
18: end function
```

```
19: function addIntervalNode (z, new)
20: if z == null then
21:   z = new
22: else if new.lb < z.lb then /* left subtree */
23:   if new.sl == z.sl + (z.lb - new.lb) then /* extend z */
24:     /* new is superinterval of z */
25:     z.lb = new.lb
26:     z.rb = min(z.rb, new.rb)
27:     z.sl = new.sl
28:     z.S_sl = new.S_sl /* new.S_sl is subset of z.S_sl */
29:     if z.left != null then
30:       z.left.rb = min(z.left.rb, z.lb - 1)
31:       /* rb of subtree nodes are fixed by updateITree */
32:   else /* add new to interval-tree */
33:     new.rb = min(z.lb - 1, new.rb)
34:     if z.left == null then
35:       z.left = new
36:     else
37:       z.left = addIntervalNode(z.left, new)
38: else if new.lb > p.lb then /* right subtree */
39:   /* z is not superinterval of new */
40:   if (z.sl != new.sl + new.lb - z.lb) then
41:     /* z.rb must be < any lb in its right subtree*/
42:     z.rb = min(z.rb, new.lb - 1)
43:     if z.right == null then
44:       z.right = new
45:     else
46:       z.right = addIntervalNode(z.right, new)
47: end function

48: function updateITree (z, maxRB)
49: if z != null then
50:   updateITree(z.left, z.lb - 1)
51:   if maxRB != -1 then
```

```
52:      z.rb = min(z.rb, maxRB)
53:   updateITree(z.right, maxRB)
54: end function
```

Figure 3-3. Algorithm 2 - Construction of an interval tree from a suffix tree.

Example 3-1

Let T be the generalized suffix tree constructed of Q_1 and $\mathbf{S} = \{S_1, S_2, S_3\}$ (Figure 3-4a). Branch nodes of T are represented by red circles. Terminal nodes of T are represented by gray rectangles. T is traversed bottom-up.

Algorithm 2 starts by calling function `traverse`, with a root node of T (v_6) as argument. Then, at each branch node v of T ($v = v_1, .., v_6$), its terminal nodes are examined. If a terminal node w is a query terminal node, then a new interval-node z is formed by calling function `addIntervalNode`. An interval-node z has following fields: lb, rb, sl, $\mathbf{S}_{sl}$, $left$ and $right$, as previously explained, and is represented in the following form in Figure 3-4b: $[lb, rb]$ $\mathbf{S}_{sl}$, sl. In the subsequent sections this is abbreviated to as $[lb, rb]$.

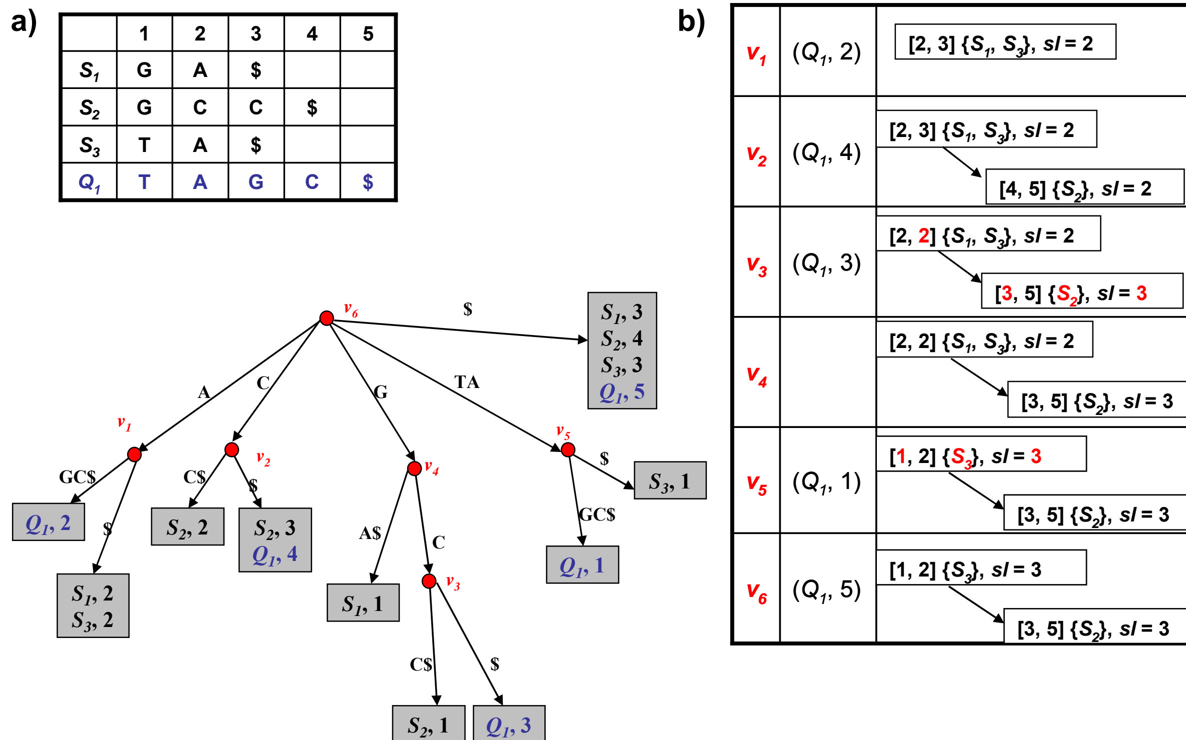


Figure 3-4 Algorithm 2 – Example 3-1. a) The generalized suffix tree *T* is built from the set of four sequences $\{Q_1, S_1, S_2, S_3\}$ listed in the top-left corner of the figure. Branch nodes of *T* are shown in red, and terminal nodes as gray rectangles designated as (S_i, p) for subject terminal nodes, and (Q_1, p) for query terminal nodes. **b)** State of the interval tree IT_1 after each of the branch nodes of *T* $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ is encountered during the bottom-up traversal of *T* (the right-most column of the table). Algorithm 2 starts by calling the function `traverse` for the root node (v_6). The function `formIntervalNode` adds a new interval node, or modifies the existing one for each query terminal child of v_i. Each interval node is represented as: [*lb*, *rb*] $\boldsymbol{S_{sl}}$, *sl*, where *lb* and *rb* are the left and the right endpoints of an interval; *sl* is the shustring length at the beginning of the interval, and $\boldsymbol{S_{sl}}$ is the set of subject sequences with the highest shustring lengths across that interval. Function `min` returns the minimum value of the specified parameters.

The construction of the interval-tree, IT_1, from Q_1 and $\mathbf{S} = \{S_1, S_2, S_3\}$ starts with the branch node v_1. The node v_1 has 3 child nodes. Two subject terminal nodes of v_1 are $(S_1, 2)$ and $(S_3, 2)$. Thus, the list $v_1.subjectId$ contains S_1 and S_3. A query terminal node of v_1 is $(Q_1, 2)$, so the first interval-node of IT_1, $z_{1,2}$, is formed from $(Q_1, 2)$. The shulen value of $(Q_1, 2)$ is the string depth of its parent node, v_1, plus 1. The endpoints of $z_{1,2}$ are initially set to $lb = 2$, and $rb = lb + sl - 1 = 2 + 2 - 1 = 3$, thus forming the interval [2, 3]. Since $v_1.subjectId = \{S_1, S_3\}$, it follows that $z_{1,2}.\mathbf{S}_{sl} = \{S_1, S_3\}$. In summary, $z_{1,2}$, which becomes the root of IT_1, has the following structure: [2, 3] $\{S_1, S_3\}$, $sl = 2$ (see Figure 3-4b).

The next node in the traversal of T is v_2, with its terminal node $(Q_1, 4)$. The new interval-node, $z_{1,4}$, is formed as: [4, 5], $\mathbf{S}_{sl} = \{S_2\}$, $sl = 2$. The root interval-node $z_{1,2}$ = [2, 3] precedes $z_{1,4}$ = [4, 5], so the new interval is added to the right subtree of the root (see Algorithm 2, lines 38-46).

The branch node v_3 has a query terminal node, $(Q_1, 3)$. The new interval, $z_{1,3}$ = [3, 5], is the superinterval of $z_{1,4}$ = [4, 5] (see Algorithm 2, lines 22-30). In this case, a new interval-node is not added to the tree, but the previous interval, $z_{1,4}$ = [4, 5], is extended to the left. Hence, the new node $z_{1,3}$ replaces $z_{1,4}$. Further, the list of subjects $z_{1,4}.\mathbf{S}_{sl}$ is replaced by $z_{1,3}.\mathbf{S}_{sl}$, which is equal to $v_3.subjectId$. Since $v_3.subjectId \subseteq z_{1,4}.\mathbf{S}_{sl}$, then only the subjects from $v_3.subjectId$ have the maximal shulen values across all positions in the extended interval [3, 5], so $z_{1,3}.\mathbf{S}_{sl} = v_3.subjectId$.

The branch node v_4 has no query terminal children, and no changes are made to the interval tree.

Similarly to the situation for the branch node v_3, when the branch node v_5 is encountered, a new interval is not added to the tree, but the existing interval $z_{1,2}$ = [2, 2] is extended to the left, and it becomes $z_{1,1}$ = [1, 2]. In addition, the new subject list becomes the subset of the old one, so $z_{1,1}.\mathbf{S}_{sl} = v_5.subjectId = \{S_3\}$.

The last query terminal node, $(Q_1, 5)$, is a child node of the root of T, v_6. Here, no change has been made in IT_1, since $(Q_1, 5)$ is already included in the interval node [3, 5], that is, $z_{1,5}$ is the subinterval of $z_{1,3}$.

The process of adding an interval in the left subtree of an interval-node is similar to the process of adding an interval to the right subtree just described (Algorithm 2, lines 21-36). However, there is one difference. Let z denote an existing interval-node, and let a new interval *new* extend z to the left (see Algorithm 2, lines 31-36). Here, not only the right endpoint of extended z has to be adjusted, but also, possibly, the right endpoints of some other nodes in the subtree of z. This is done only once by calling the function `updateITree` after all interval-nodes have been added to the tree. The function `updateITree` takes O(l) time to traverse an interval-tree and fixes all right endpoints. In comparison, if the right endpoints of interval-nodes were adjusted during the insertion of every new interval-node, then the total procedure for all inserted interval-nodes would take O(l log l) time.

Further, a branch node v may contain only query terminal nodes in its subtree, i.e. v may have an empty list of subjects (this situation is not shown in Figure 3-4). In that case, interval-nodes cannot be formed from the query terminal nodes of v. These unresolved query terminal nodes are stored in the set *v.unresolvedTerm*. The set of unresolved query terminal nodes is passed from v to its ancestral branch nodes, until a node w is encountered, which is the lowest common ancestor of v and at least one subject terminal node. When w is encountered, interval-nodes can be formed of query terminal nodes stored in *v.unresolvedTerm*.

3.2.5. Computing a list of segements G_i

A segment list G_i is formed from an interval list I_i. Every segment $G_{i,d}$ of Q_i (d =1, .., k) is said to be most closely related to $\boldsymbol{S_{i,d}} \subseteq \boldsymbol{S}$, if the sum of shulens $sum_{i,d}$ over $G_{i,d}$ is maximal for the shustrings of Q_i when compared to $S_j \in \boldsymbol{S_{i,d}}$:

(i) $maxsum_{i,d} = \max \{sum_{i,d}(S_j) \mid S_j \in \boldsymbol{S}\}$,

and $sum_{i,d}(S_j) = \sum_{p=G_{i,d}.lb}^{G_{i,d}.rb} h_{i,j,p}$ for a query Q_i when compared to S_j

(ii) $\boldsymbol{S_{i,d}} = \{ S_j \in \boldsymbol{S} \mid sum_{i,d}(S_j) = maxsum_{i,d}\}$

End-points of a segment $G_{i,d}$ are determined by sliding window-analysis. A sliding window[10] $w_{i,e}$ represents a segment of Q_i of a predefined length. Subjects which are locally homologous to Q_i over $w_{i,e}$ are those with the maximal value of the sum of shulens over $w_{i,e}$. Thus, $wsum_{i,e}$ is defined as the sum of shulens over a window $w_{i,e}$:

(i) $wmaxsum_{i,e} = \max \{wsum_{i,e}(S_j) \mid S_j \in \boldsymbol{S}\}$,

and $wsum_{i,e}(S_j) = \sum_{p=w_{i,e}.lb}^{w_{i,e}.rb} h_{i,j,p}$ for a query Q_i when compared to S_j

(ii) $\boldsymbol{S_{i,e}} = \{ S_j \in \boldsymbol{S} \mid wsum_{i,e}(S_j) = wmaxsum_{i,e}\}$

When two adjacent windows, $w_{i,e}$ and $w_{i,e+1}$, have different list of subjects with maximal sums of shulens over windows ($\boldsymbol{S_{i,e}} \neq \boldsymbol{S_{i,e+1}}$), then an end-point of a segment is formed as a middle point of the overlapping part of these two windows.

However, in an interval list I_i only the values $H_{i,p}$ are stored, and values $h_{i,j,p} \neq H_{i,p}$ cannot be obtained from I_i for all positions p in Q_i when compared to every subject $S_j \in \boldsymbol{S}$. In order to use as much information as possible to compute $sum_{i,d}$ and $wsum_{i,e}$, some additional information is derived from intervals in I_i. As explained in Section 3.2.4, each interval $I_{i,p}$ contains the following fields:

(i) lb – the left endpoint of the interval: $lb = p$

(ii) rb – the right endpoint of the interval; initially: $rb = lb + H_{i,p} - 1$

(iii) sl – shulen at the beginning of the interval: $sl = H_{i,p}$

(iv) $\boldsymbol{S_{sl}}$ – a subset of $\boldsymbol{S}$, such that $\boldsymbol{S_{sl}} = \{S_j \in \boldsymbol{S} \mid |h_{i,j,p}| = H_{i,p}\}$

The window size w can be set by a user, or a default value (1 nucleotide) is used. The windows are advanced by an increment that is also either set by the user, or defaults to the window size divided by 10. Further, the minimal fragment length f is

[10] Let w be the window size, and inc the increment step in a sliding windows analysis. The first sliding window $w_{i,1}$ represents the substring $Q_i[0 .. w - 1]$, the second window $w_{i,2}$ represents $Q_i[inc .. inc + w - 1]$, etc.

introduced as the minimal length of the segment that can be considered as a relevant match between a subject and a query. The value f can be also set by the user, or its default value is the length of an increment step. If a segment length is shorter than f, that segment is discarded, and two neighboring segments are fused across this segment.

3.3. Analysis of *st* on simulated data sets

3.3.1. Run-time and memory usage analysis of *st*

Run-time and memory usage analysis were computed for simulated data sets of nucleotide sequences of length (l) 10 kb, 100 kb and 1 Mb. Samples were simulated using the program Dawg (Cartwright, 2005). Each sample consists of a query sequence, and a set of subject sequences. In experiments (i) and (ii), the subject set contains n = 10 sequences, and in experiment (iii) n = 100 sequences. The summary characteristics of these three experiments are shown in Table 3-1. Notice that a query sequence is generated as a different recombinant in each of experiments:

Experiment (i) (n = 10): a query is a recombinant of subject sequences 1 and 2 in the following order: 1-2-1-2-1, i.e. the first segment is most closely related to subject 1, the second segment is most closely related to subject 2, and so on. Thus, for the data sets of l = 10 kb and l = 100 kb, a query sequence consists of 5 segments of equal lengths (2 kb and 20 kb, respectively), and of 50 segments for the data sets of l = 1 Mb. Notice that for the data sets of l = 1 Mb, the mosaic 1-2-1-2-1 is repeated 10 times to construct 50 segments.

Experiment (ii) (n = 10): a query is a recombinant of subject sequences 1, 2, 6, and 9 in the following order: 1-2-6-1-9, i.e. the first segment is most closely related to subject 1, the second segment is most closely related to subject 2, the third segment is most closely related to subject 6, and so on. Again, for the data sets of l = 10 kb and l = 100 kb, a query sequence consists of 5 segments of equal length (2 kb and 20 kb, respectively), and of 50 segments for the data sets of l = 1 Mb. The mosaic 1-2-6-1-9 is repeated 10 times to construct 50 segments in the case of the data sets of l = 1 Mb.

Experiment (iii) (n = 100): a query is a recombinant of subject sequences 1 and 2, in the following order: 1-2-1-2-1 as in Experiment (i). As before, for the data sets of l = 10 kb and l = 100 kb, a query sequence consists of 5 segments of equal length (2 kb and 20 kb respectively), and of 50 segments for the data sets of l = 1 Mb. As in Experiment (i), the mosaic 1-2-1-2-1 is repeated 10 times to construct 50 segments for the data sets of l = 1 Mb.

Table 3-1. Run-time and memory usage analysis of *st* on simulated data sets. For each of three experiments, three data sets of different sequence length (l) were generated. In experiments (i) and (ii), the set of subjects contains n = 10 sequences. In experiment (iii), n = 100. For each experiment, the mosaic form (*mosaic*) of a recombinant sequence (query) is shown. The table contains two statistics for each experiment: the run-time required for the execution of *st* (t), and the memory usage peak of *st* (m).

	Experiment (i) **n = 10** ***mosaic*: 1-2-1-2-1**		**Experiment (ii)** **n = 10** ***mosaic*: 1-2-6-2-9**		**Experiment (iii)** **n = 100** ***mosaic*: 1-2-1-2-1**	
	m [MB]	**t [s]**	**m [MB]**	**t [s]**	**m [MB]**	**t [s]**
l = 10 kb	2.9	0.072	2.8	0.072	23.4	1.874
l = 100 kb	29.0	2.088	28.9	2.083	233.9	37.839
l = 1 Mb	293.8	39.358	292.96	39.274	2344.3	482.188

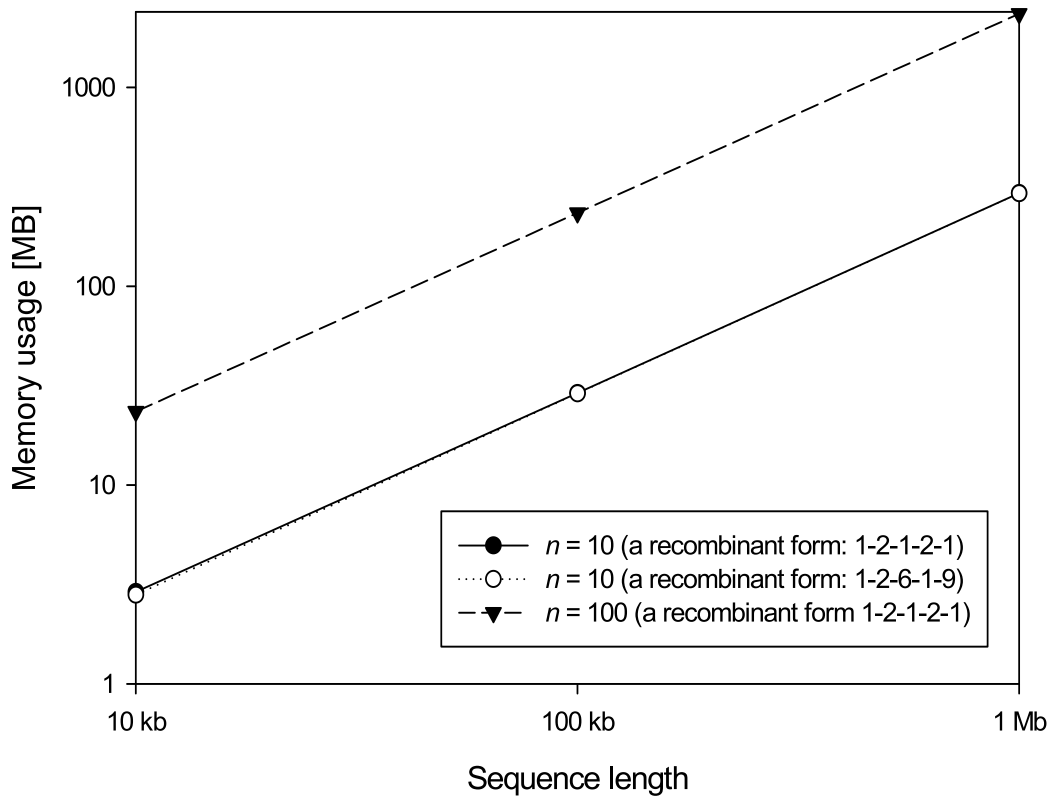


Figure 3-5. Memory usage of *st* on simulated data sets. The analysis shows that the memory usage is linear in the size of the input data set.

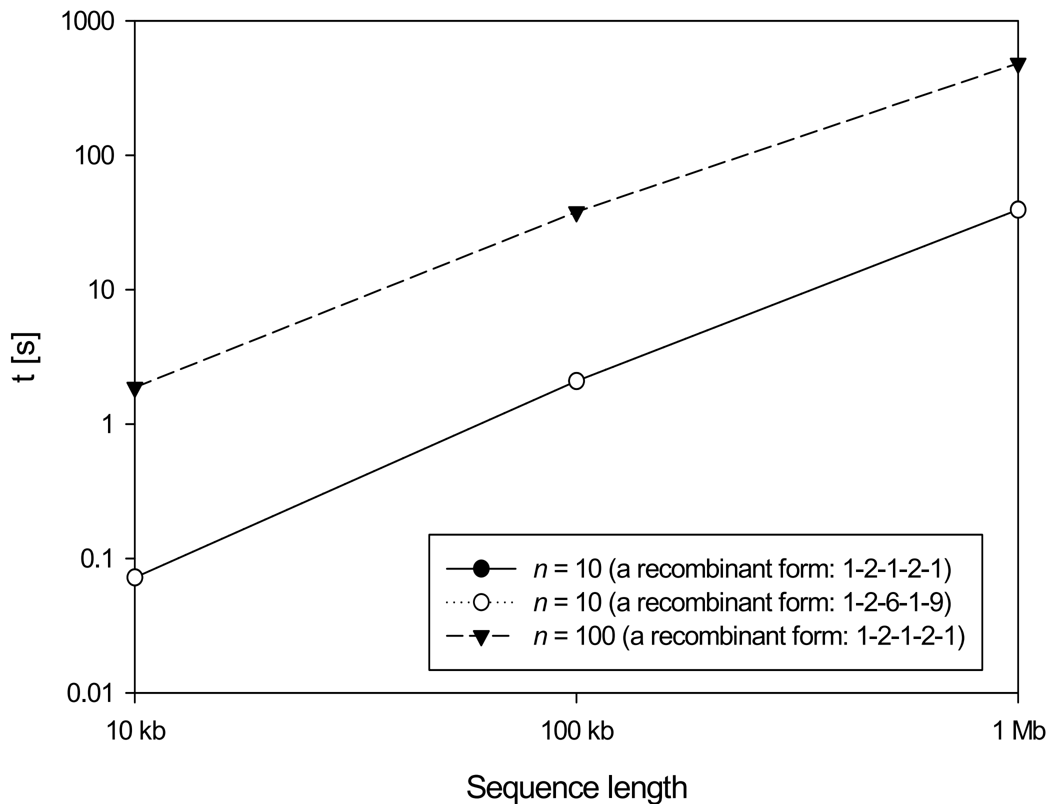


Figure 3-6. Run-time of *st* on simulated data sets. Three graphs are shown, each representing a different combination of the number of subject sequences in the data set (*n*), and a mosaic form representing the recombinant query sequence.

3.3.2. Consistency of *st*

In this Section, the consistency of my program, *st*, is tested on simulated data sets. The analysis is based on the samples simulated by Dawg (Cartwright, 2005). In all three experiments, the model and the parameters used were as set by Kosakovsky Pond *et al*. (2009) in order to simulate HIV-1 scenario: the general time reversible model of nucleotide substitution with gamma and invariant rate heterogeneity (the shape parameter of $\alpha = 0.8$); the four stationary nucleotide frequencies are $\pi_A = 0.4$, $\pi_C = 0.2$, $\pi_G = 0.1$, and $\pi_T = 0.3$; and the six substitution rates are $\theta_{AC} = 2.0$, $\theta_{AG} = 4.0$, $\theta_{AT} = 0.8$, $\theta_{GC} = 0.9$, $\theta_{CT} = 5.0$, and $\theta_{GT} = 1.0$.

In Experiment 1, 100 samples of 4 sequences of length 10 kb were simulated. In each data set, three sequences represent subject sequences (sequences denoted 1, 2, and 3), and one sequence represents a query sequence (denoted Q), which is constructed as a recombinant of sequences 1 and 2. More precisely, a query sequence contains 5 fragments of length 2 kb, where each fragment is most closely related to sequence 1 or sequence 2, in alternation (see Figure 3-7). Parameter d in Figure 3-7 represents the length of the shortest branch in a model, and is scaled as $2d$ and $3d$ in other branches. Moreover, the length of each branch in the model is the expected number of the substitutions per site. For example, the first fragment of Q is the most closely related to sequence 1, and its evolutionary distance to sequence 1 is 0.01 + 0.01 = 0.02. The value of d ranges from 0.001 to 0.25 in order to analyze the accuracy of *st* over different substitution rates (see Figure 3-7). In summary, there are two phylogenies that involve a recombinant sequence Q. The first phylogeny refers to the fragments of Q which are locally homologous to 1 and it is (((Q:d, 1:d): d, 2:2d): d, 3:3d); and the second phylogeny refers to the fragments of Q which are most closely related to 2: (((Q:d, 2:d): d, 1:2d): d, 3:3d);

Figure 3-8 contains three graphs, each corresponding to the analysis based on a different window length (w = 600 bp, 1 kb, and 2 kb). Each point on the graph represents the mean accuracy +/- standard deviation of recombination detection over 100 samples. The accuracy can range from 0 to 1, and it refers to the ratio of correctly classified nucleotide positions in a query sequence. For example, let Q be constructed in the form 1-2-1-2-1, where each fragment is of length 2 kb, as in Experiment 1. If Q is then predicted by *st* to be most closely related to the sequence 1 over its whole length, then the prediction accuracy is 0.6, since 6 kb out of 10 kb were correctly determined, and 4 kb (which were most closely related to the sequence 2) were misclassified.

In Experiment 2, 100 samples of 4 sequences of length 100 kb were simulated. A query sequence in each sample is constructed as a recombinant of sequences 1 and 2 and contains 50 fragments of length 2 kb, where the pattern 1-2-1-2-1 is repeated 10 times (Figure 3-9). Again, the windows analysis (w = 600 bp, 1 kb, 2 kb, 10 kb) is applied to see the prediction accuracy of this more complicated model over a range of d values.

Finally, in Experiment 3, 100 samples of 4 sequences of length 100 kb were simulated, where a query sequence contains 5 fragments of length 20 kb in the recombination form 1-2-1-2-1 (Figure 3-10). Windows analysis (w = 600 bp, 1 kb, 2 kb, 10 kb, 20 kb) is applied over a range of d.

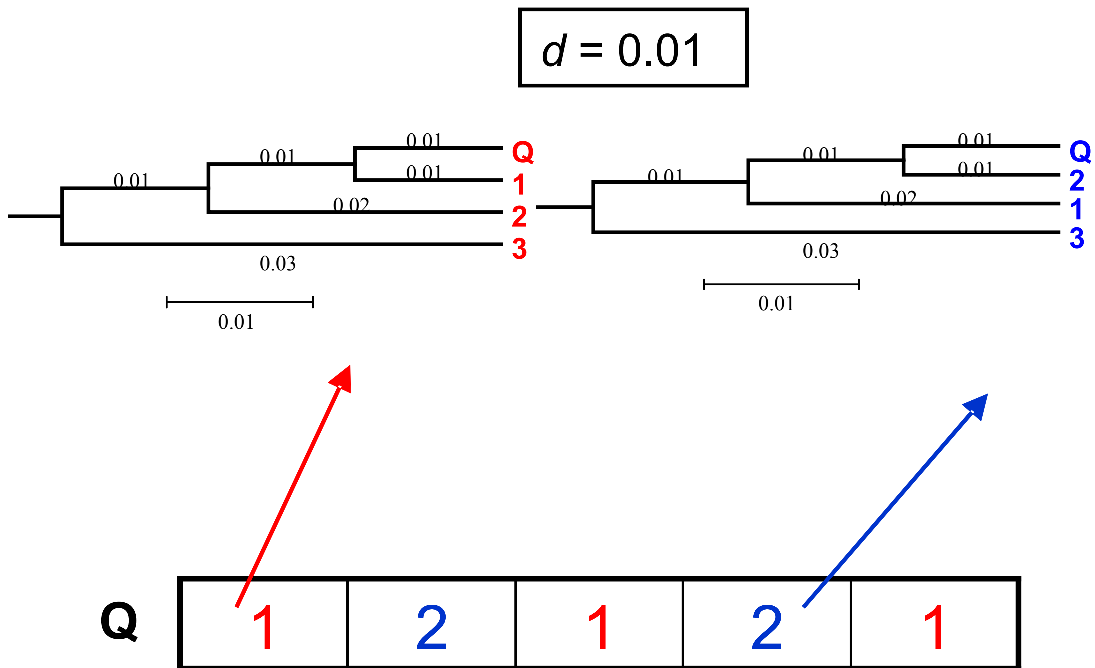


Figure 3-7. Phylogeny of simulated recombinants. In each of three scenarios, used in Experiments 1-3, 4 sequences were constructed: three subject sequences (denoted 1, 2, and 3), and a query sequence (denoted Q). A recombinant query Q is constructed as a concatenation of either 5 fragments (in Experiments 1 and 3), or of 50 fragments (in Experiment 2). Each fragment is most closely related to either a subject sequence 1 or a subject sequence 2. Here, 5 fragments of Q are drawn, each corresponding to either the sequence 1 (a fragment denoted by 1), or to the sequence 2 (a fragment denoted by 2). In Experiments 1 and 3, a recombinant is of the form 1-2-1-2-1, and in Experiment 2, these form is repeated 10 times, as there are 50 fragments in a recombinant. A parameter d = 0.001, 0.0025, 0.005, 0.01, 0.05, 0.1, 0.2, 0.25, which is the shortest branch in both shown phylogenies, also represents the expected number of substitutions per site. For example, since here d is 0.01, then the left phylogenetic tree is depicted as (((Q:0.01, 1:0.01): 0.01, 2:0.02): 0.01, 3:0.03);

The following conclusions can be derived from the analysis in Experiments 1-3, as seen in Figures 3-8 to 3-11:

(i) as the fragment length growths, the detection accuracy is better; for example, for d = 0.001, the average accuracy in Figure 3-8 is around 0.7 (a recombinant sequence is formed of 5 fragments of 2 kb), and in Figure 3-10 is around 0.8 (a recombinant sequence is formed of 5 fragments of 20 kb)

(ii) the window size, w, only slightly influences the overall prediction accuracy as long as w is not longer than a recombinant fragment length; when w is greater than the fragment length, the prediction accuracy starts declining. For example, in Figure 3-9 (a recombinant sequence is formed of 50 fragments of 2 kb) the accuracy is consistent for $w \leq 2$ kb, but significantly worse results are obtained for $w = 10$ kb.

(iii) with the constant window size, the number of fragments in a recombinant only slightly reduces the accuracy; for example, compare the results in Figures 3-8 (a recombinant sequence is formed of 5 fragments of 2 kb), and in Figure 3-9 (a recombinant sequence is formed of 50 fragments of 2 kb)

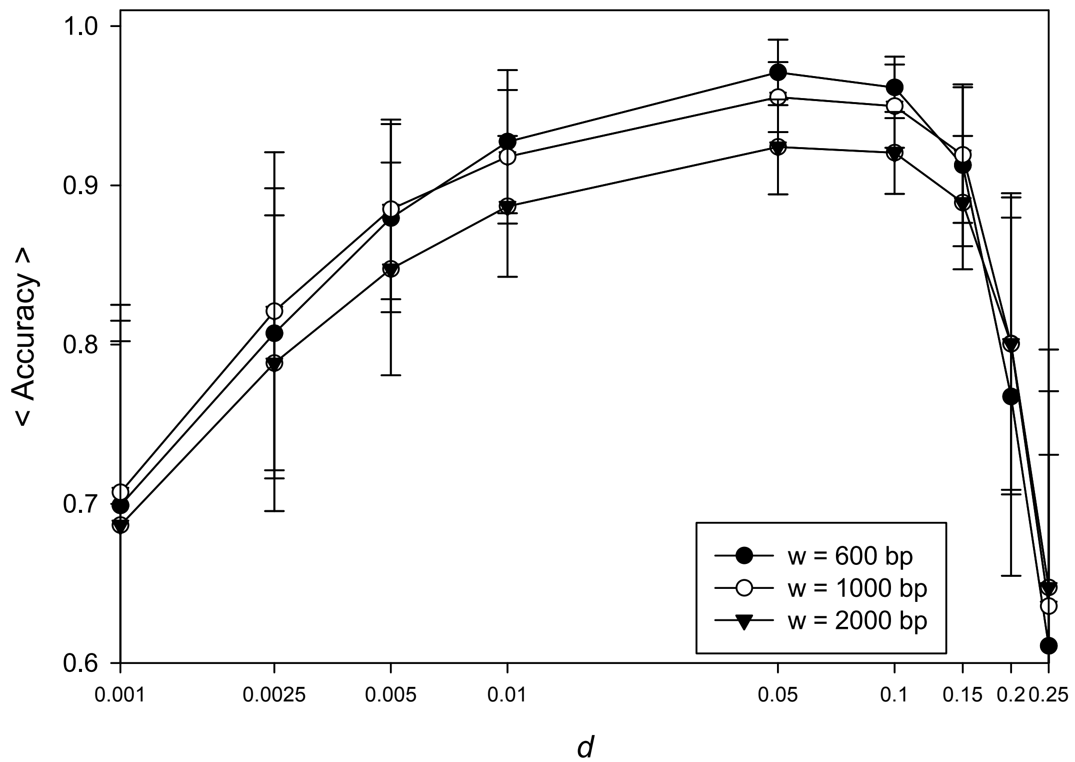


Figure 3-8. Recombination detection accuracy for data samples of sequences of length 10 kb, and a recombinant consisting of 5 fragments of length 2 kb. Figure contains 3 graphs, each for a different window length (*w*). Each point represents a mean value +/- SD of 100 samples.

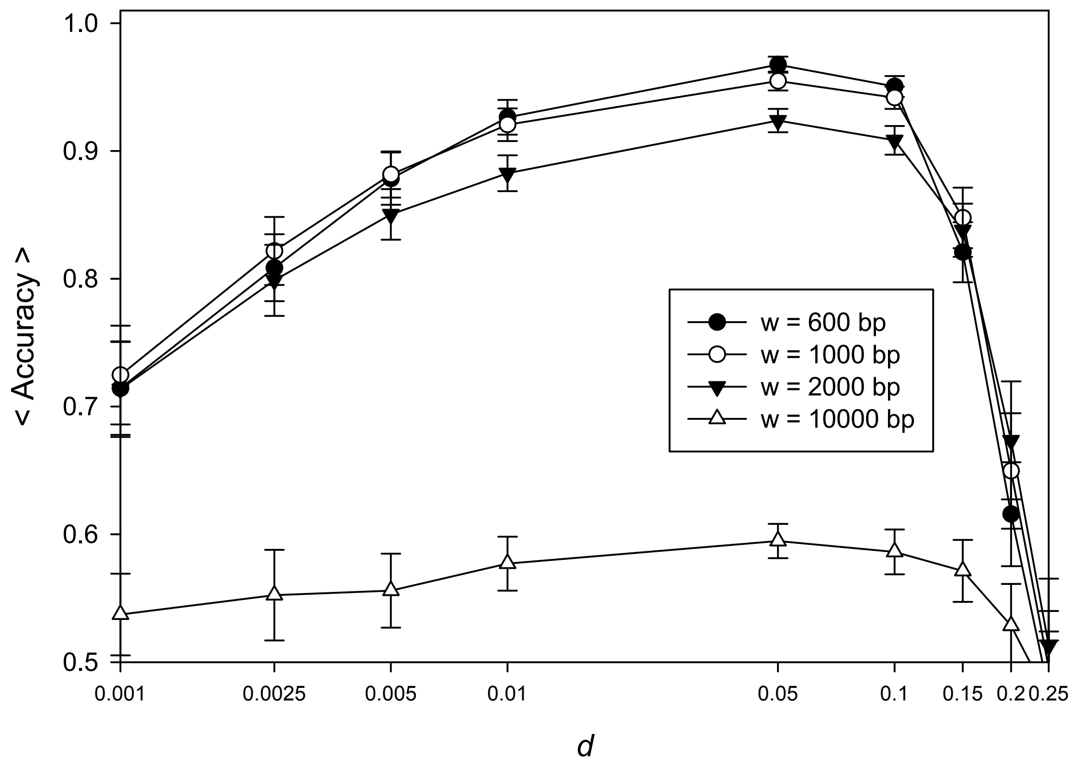


Figure 3-9. Recombination detection accuracy for data samples of sequences of length 100 kb, and a recombinant consisting of 50 fragments of length 2 kb. Figure contains 4 graphs, each for a different window length (*w*). Each point represents a mean value +/- SD of 100 samples.

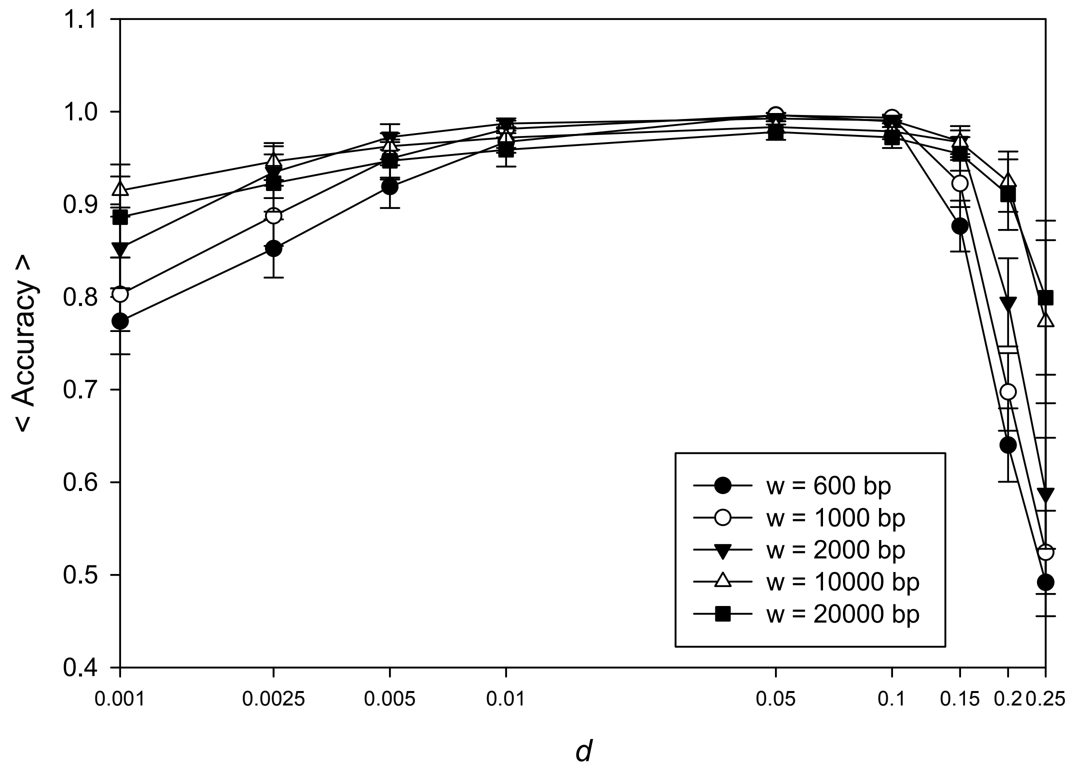


Figure 3-10. Recombination detection accuracy for data samples of sequences of length 100 kb, and a recombinant consisting of 5 fragments of length 20 kb. Figure contains 5 graphs, each for a different window length (*w*). Each point represents a mean value +/- SD of 100 samples.

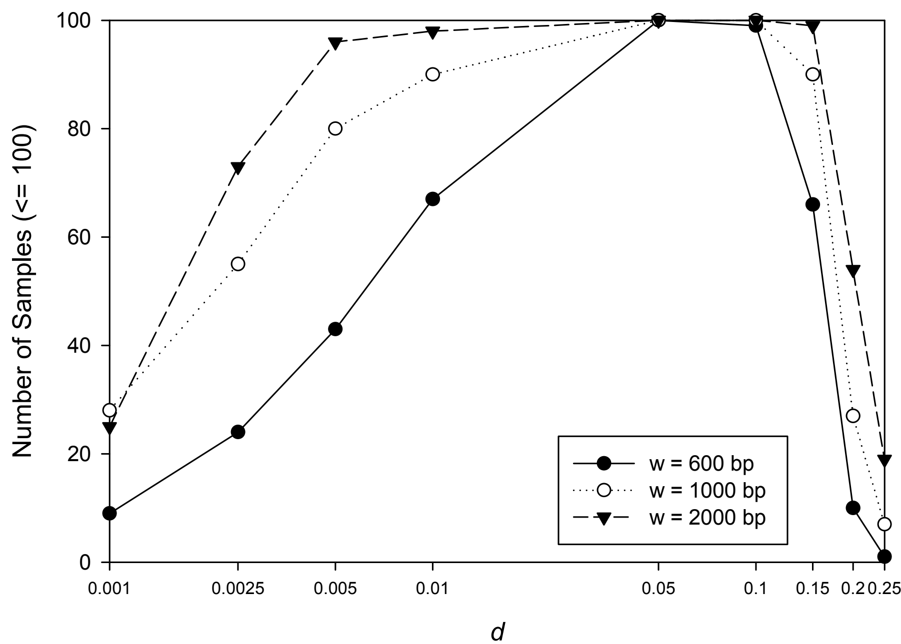


Figure 3-11. Recombination detection accuracy for data samples of sequences of length 10 kb, and a recombinant consisting of 5 fragments of length 2 kb. Figure contains 5 graphs, each for a different window length (*w*). Each point represents the number of samples (out of generated 100) which were completely correctly classified. That is, all 4 recombinant breakpoints were correctly classified, and the correct parental sequences were determined for both sides of each breakpoint.

3.3.3. Comparison to SCUEAL on simulated data sets

Finally, the recombinant detection performance of *st* is compared to SCUEAL (Kosakovsky Pond *et al.*, 2009), since this tool was specifically designed for the highly precise analysis of HIV sequence. The authors extensively analyzed SCUEAL over both real and simulated data sets, and the availability of these data together with the results was another reason for choosing this program as the accuracy standard. However, as a trade-off, SCUEAL implementation is time consuming, and thus impractical for very large sequences (e.g. SCUEAL analysis of an HIV recombinant lasted more than 6 hours, while *st* analysis of the same sequence took 0.4 seconds; Section 3.4.2). Therefore, in this Section, I only concentrate on the accuracy

performance of *st* in comparison to SCUEAL on simulated data sets provided by Kosakovsky Pond *et al*. (2009).

The simulated data sets used in the experiments resemble the simple and the complex HIV-like mosaic. In each experiment, 100 samples were generated, where each sequence is 10 kb long, with the recombinant fragment length of 2 kb (Kosakovsky Pond *et al*., 2009). For each sample, a recombinant query sequence is compared to 11 subject sequences (sequences denoted 1 to 11).

In the simple mosaic scenario, a query sequence is constructed in three forms: 1-2-1-2-1, 1-6-1-6-1, and 1-9-1-9-1. In the first case, evolutionary distance between sequences 1 and 2 is close; in the second case, evolutionary distance between sequences 1 and 6 is medium, and in the last case, sequences 1 and 9 are divergent (Kosakovsky Pond *et al*., 2009). In the complex mosaic scenario, the recombinant form is 1-2-6-1-9, and the subject sequences are close (sequence 2), medium (sequence 6), or distantly-diverged (sequence 9) from the sequence 1. The results of SCUEAL-based analysis on these data sets (Kosakovsky Pond *et al*., 2009) are presented in Table 3-2.

Table 3-2. Results of SCUEAL analysis of HIV-like recombinant detection (Kosakovsky Pond *et al*., 2009).

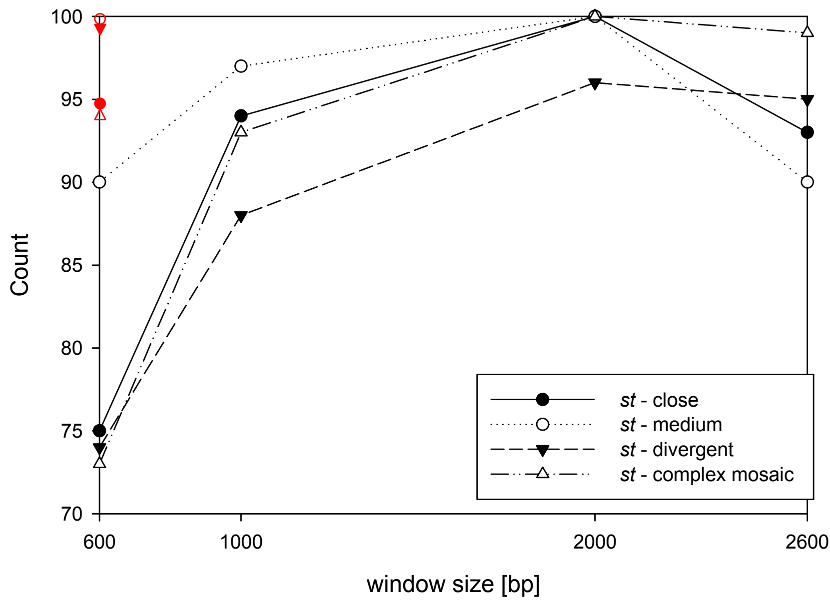
SCUEAL	***close***	***medium***	***divergent***	***complex mosaic***
Correct	95/100	100/100	99/100	94/100
Subset	1/100	0	0	5/100
Superset	2/100	0	1/100	0
Mismatch	2/100	0	0	1/100

The summary results of *st* analysis and the comparison to SCUEAL are shown in Figures 3-12 and 3-13. The analysis shows that the window size, *w*, close to the fragment length gives better results, but the results obtained from the windows analysis with *w* up to the half of the fragment length (for *w* = 1kb) are also acceptable. These results agree with the previous analysis (see Section 3.4.2). In addition, results with the greater value of *f* (*f* is the minimal length of *st* fragment to

be considered as a relevant), are better as long as window size does not exceed the true fragment length.

Further, the best overall results are obtained for the simple HIV-mosaic scenario, where the parental sequences of a recombinant are at medium evolutionary distance (mosaic form: 1-6-1-6-1). The only case where *st* outperforms SCUEAL is the complex HIV mosaic scenario for window size at least 1 kb. In all other cases, *st* is surpassed by SCUEAL results. With a proper choice of the window size (*w*), and the minimal fragment length (*f*), *st* results are closely matching those obtained by the SCUEAL analysis (Figures 3-12 and 3-13).

a) *st* results compared to SCUEAL results

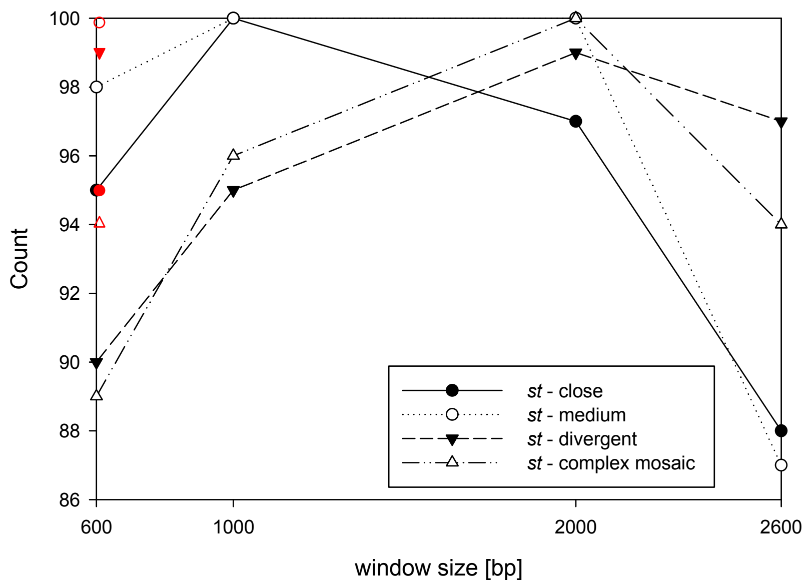


b) SCUEAL results

close	medium	divergent	complex mosaic
95	100	99	94

Figure 3-12. Comparison of *st* and SCUEAL results, where the minimal fragment length (*f*) of *st* was set to 200 bp. These programs were compared over 4 different scenarios (see text). The *st* windows analysis is shown in for a range of window size (600 bp, 1 kb, 2 kb, 2.6 kb) (black graphs). The value *Count* on y-axis refers to the number of samples (out of 100) for which all 4 breakpoints in a recombinant were correctly recognized (the position, and the parental sequences). SCUEAL results are shown in b) (see also Table 3-2), and denoted as red circles, and red triangles in a).

a) *st* results compared to SCUEAL results



b) SCUEAL based results

close	medium	divergent	complex mosaic
95	100	99	94

Figure 3-13. Comparison of *st* and SCUEAL results, where the minimal fragment length (*f*) of *st* was set to 400 bp. The programs were compared over 4 different scenarios (see text). The size of a window in *st* windows analysis ranges from 600 bp to 2.6 kb (black graphs). The value *Count* on y-axis refers to the number of samples (out of 100) for which all 4 breakpoints in a recombinant were correctly recognized. SCUEAL results are shown in b) (see also Table 3-2), and denoted as red circles, and red triangles in a).

3.4. Application of *st*

The program *st* was used to detect local sequence similarity in real data sets. The results are compared to the results of existing tools, and where possible, to the published annotation. The following data sets were analyzed (Table 3-3):

(i) *Neisseria meningitidis* in comparison to *N. gonorrhoeae* and *N. cinerea*

(ii) HIV-1 strain A_DQ083238 in comparison to 37 reference strains from the HIV sequence database (http://www.hiv.lanl.gov)

(iii) 91 HIV-1 circulating recombinant forms in comparison to HIV-1 42 pure subtype strains (Leitner *et al*., 2005; Wu *et al*., 2007)

(iv) 266 HIV-1 circulating recombinant forms in comparison to HIV-1 42 pure subtype strains, and 65 recombinant strains (Leitner *et al*., 2005; Wu *et al*., 2007)

(v) avian pathogenic *E. coli* strain O1:K1:H7 (Johnson *et al*., 2007) in comparison to human pathogenic *E. coli* genomes

Table 3-3. Analyzed data sets. The table contains following statistics for each data set: the number of subject sequences in the data set (*n*), the number of query sequences in the data set (*m*), the average sequence length (l_{avg}), the size of the whole data set (*size*), the reference for the analysis of the data set (*Compared to*), the time required for the execution of *st* (t_{st}), *st* memory usage peak (m_{st}).

Data set	***n***	***m***	l_{avg} **[kb]**	**size [kb]**	**Compared to**	t_{st} **[s]**	m_{st} **[MB]**
Neisseria	2	1	0.9	2.7	Boni *et al*. 2007; Westesson and Holmes, 2009	0.005	0.1
HIV	37	1	9	342	SCUEAL (Kosakovsky Pond *et al*. 2009)	0.4	8
HIV	42	91	9	1169	Wu *et al*. 2007; NCBI genotyping tool (Rozanov *et al*., 2004)	4	43
HIV	42+65	266	9	3329	Wu *et al*. 2007; NCBI genotyping tool (Rozanov *et al*., 2004)	23	112
E. coli	13	1	4756	66588	Johnson *et al*. 2007	325	1582

3.4.1. The analysis of *Neisseria meningitidis*

The first data set analyzed consists of sequences representing argF gene of three species from the genus *Neisseria*: *N. meningitidis*, *N. cinerea*, and *N. gonorrhoeae*. Previous studies have already shown the mosaic structure of argF gene of *N. meningitidis* (e.g. Boni *et al*., 2007; Westesson and Holmes, 2009). Their results suggest that the first part of argF gene of *N. meningitidis* (nucleotide positions from 1 to approximately 200) is most closely related to the *N. cinerea* argF gene, and that the ancestry of the second part of the gene is in *N. gonorrhoeae*. However, the position of another possible breakpoint (around nucleotide positions 50 or 750) remains unclear (Boni *et al*., 2007; Westesson and Holmes, 2009).

The results obtained from *st* confirm the previous results (Boni *et al*., 2007). The results shown in Figure 3-14 were based on *st* analysis using windows size: 30 bp, the increment step: 10 bp, and the minimal length of the fragment to be considered as a relevant transfer: 10 bp. The similar results were obtained with windows size varying from 30–200. However, if the size of the minimal relevant transfer is increased, then the second break point (around nucleotide position 750) is not detected.

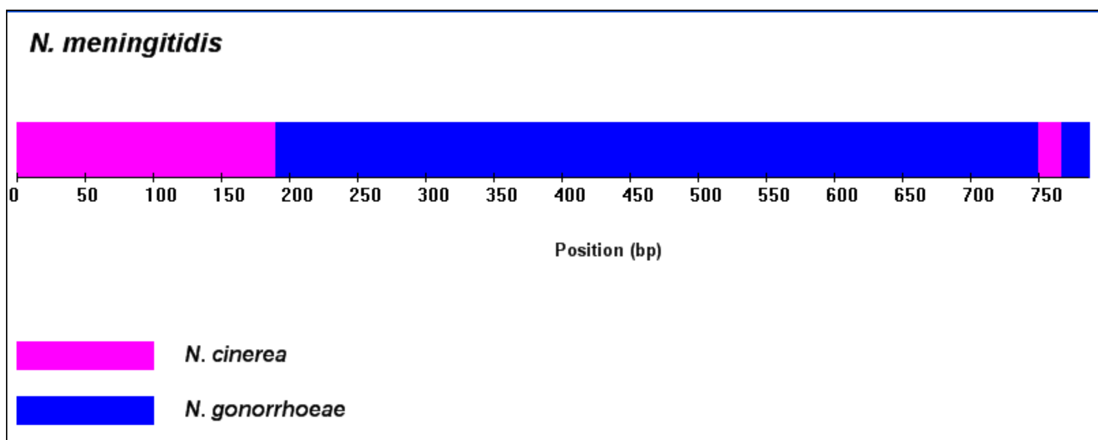


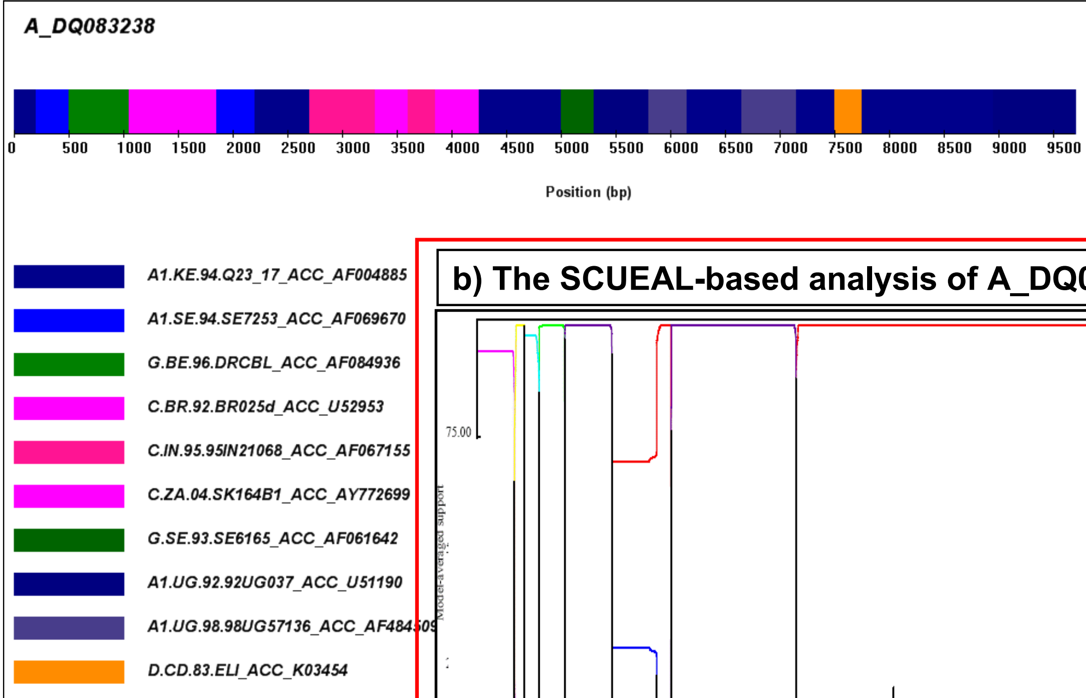
Figure 3-14. The mosaic structure of *N. meningitidis* detected by *st*. The segments of *N. meningitidis* which are locally homologous to *N. cinerea* are represented as pink rectangles (nucleotide positions 0-189, and 750-766), and segments which are locally homologous to *N. gonorrhoeae* are represented as blue rectangles (nucleotide positions 190-749, and 767-786). The recombination form of *N. meningitidis* derived from *st* analysis agrees with the previous results (e.g. Boni *et al*., 2007). However, the existence of the second breakpoint (around nucleotide position 750) remains unclear.

3.4.2. The analysis of a recombinant form of HIV-1

The next example shows the recombinant form of an HIV-1 strain, A_DQ083238, which was not classified according to its official subtype, A (see Section 2.4.4, and Figure 2-13). The strain was compared to the set of 37 HIV-1 reference strains (HIV sequence database, http://www.hiv.lanl.gov). It was analyzed both by *st*, and by SCUEAL, a recently developed phylogeny-based tool for the sequence subtyping

(Kosakovsky Pond *et al*., 2009). Both *st* and *SCUEAL* found that the ancestry of most of A_DQ083238 strain is of A1 sub-subtype, but with a significant part (around 30%) of C subtype genome (Figures 3-15 to 3-17). The *st*-based results were computed with window size 300, increment step 50, and the minimal fragment length of 100. The precise results generated by SCUEAL were produced in 379 minutes ≈ 6.3 h, and the memory usage peak was 169 MB.

The strong local sequence homology between A_DQ083238 and C strains resulted in the misclassification of A_DQ083238 in K_r-based phylogeny (the strain was clustered with C, instead of A strains). The strong signal was based on the high values of shortest unique substrings between A_DQ083238 and C strains, which resulted in higher average length of shortest unique substrings when compared to the average length of shustrings of A_DQ083238 when compared to A strains.



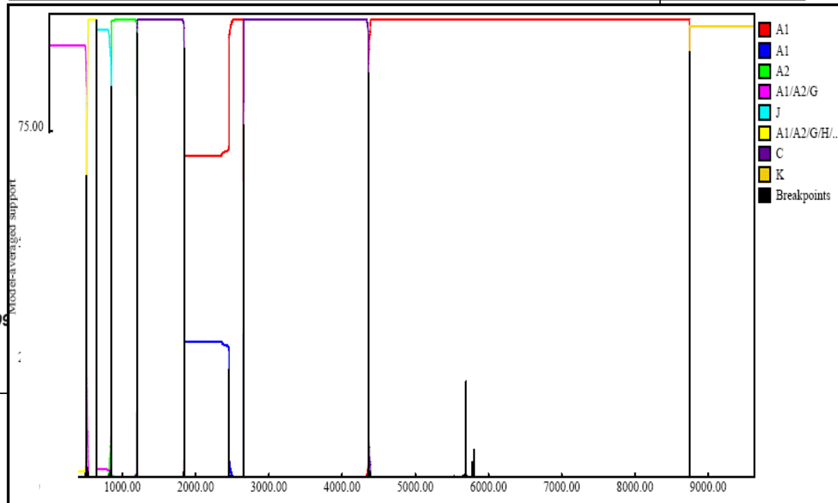


Figure 3-15. The analysis of the strain A_DQ083238 by *st* and SCUEAL. Both methods showed the recombinant form of this strain: it consist of mostly A subtype and around 30% of C subtype. In addition, both programs detected two regions of C subtype: the first regions from roughly 1000-1800 bp, and the second region from 2700-4200 bp.

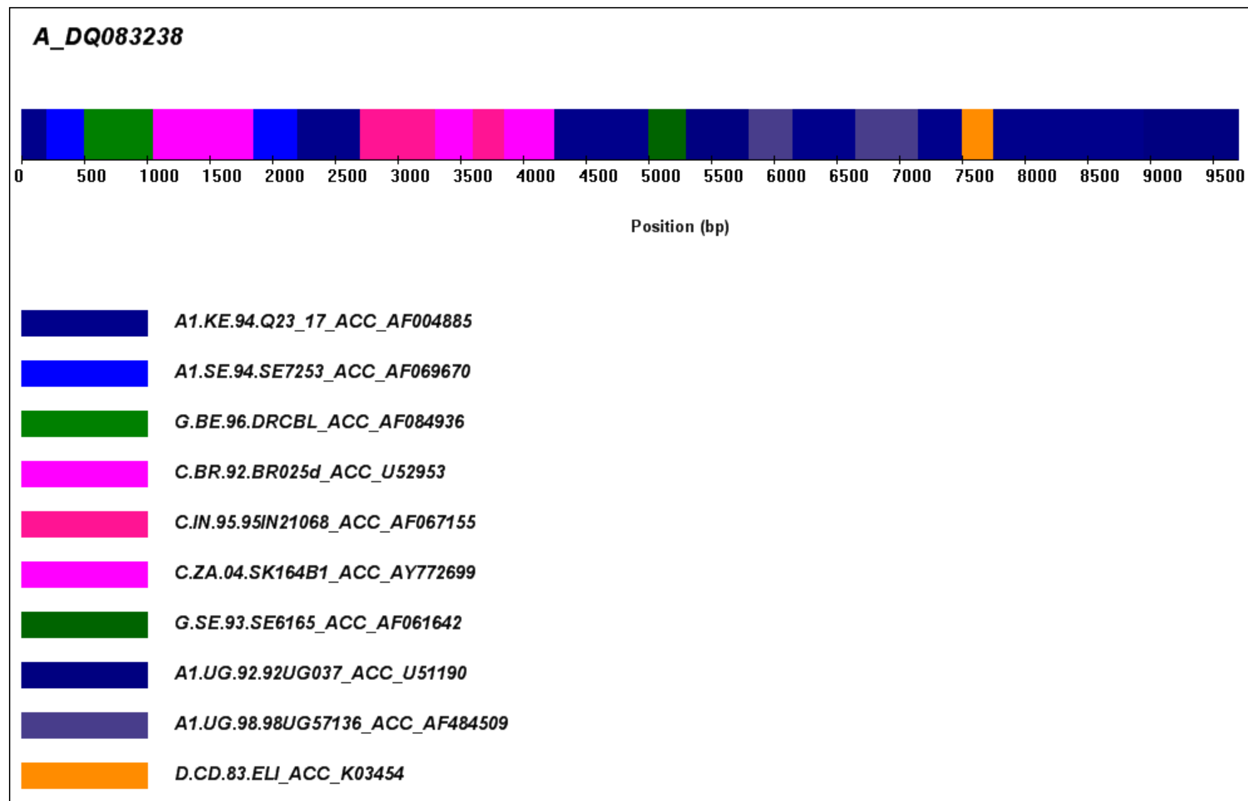


Figure 3-16. Analysis of strain A_DQ083238 by *st*. A_DQ083238 was compared to 37 reference pure subtype strains (see text). The segments of A_DQ083238 which are most locally homologous to A1 strains are represented as four shades of blue rectangles (e.g. nucleotide positions 0-499), and segments which are locally homologous to C strains are represented as three shades of pink rectangles (nucleotide positions 1050-1849, and 2700-4249). The *st* results suggest that A_DQ083238 is a recombinant form of mostly A1 sub-subtype with 30% of the genome derived from C subtype, and some traces of G subtype.

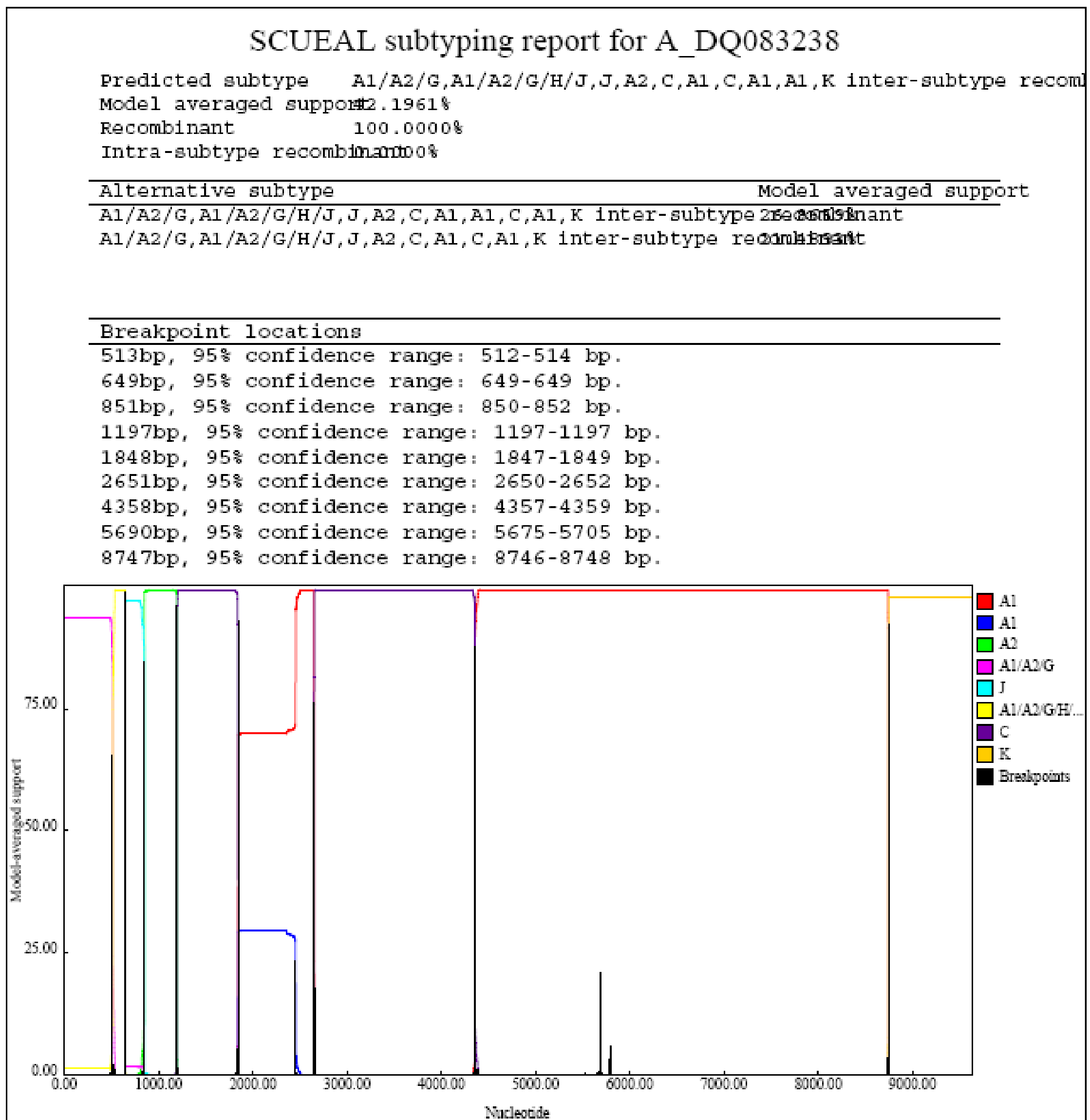


Figure 3-17. A part of the report generated by SCUEAL for strain A_DQ083238. A_DQ083238 was compared to 37 reference pure subtype strains. This phylogeny-based analysis reveals the recombinant nature of the strain, with prevailing A1 sub-subtype, and about 30% of C derived genome.

3.4.3. The analysis of circulating recombinant forms of HIV-1

Wu *et al*. (2007) analyzed 331 circulating recombinant forms (CRFs) of HIV-1. They analyzed the data set using NCBI genotyping tool, also designed for the analysis of HIV recombinant forms (Rozanov *et al*. 2004), and their solution based on the nucleotide composition of string vectors.

First, the subset of 91 recombinant strains was analyzed in comparison to the set of 42 HIV-1 reference sequences (Leitner *et al*., 2005; Wu *et al*., 2007). Each strain from this subset is officially classified as a recombinant of two subtypes, e.g. a strain classified as CRF02AG is a recombinant of A and G subtypes. The authors reported the prediction accuracy of 87.3% for their method, and 73.4% and 66.2% for NCBI genotyping tool. In the *st* analysis of these 91 strains, parental subtypes were determined as the two top subject strains returned by *st* windows analysis. The accuracy of *st* was 93.4%.

Next, from the set of 331 CRFs, 65 of them were chosen as reference recombinant forms. Thus, the remaining 266 strains were compared to the reference set of 42 pure subtypes and 65 recombinant forms to find the most closely related strain. NCBI tool correctly classified 264, and Wu *et al*. (2007) 242 strains. For this data set, *st* correctly subtyped 263 strains. Among three strains for which *st* did not return the correct most closely related strain, two strains, AY771588 and AY771589, were most closely related to a strain of pure subtype (B), which was also the result of NCBI genotyping tool reported by Wu *et al*. (2007). The closer inspection showed that the top result (a strain of subtype B) was closely followed by strains of BF recombinant form, which was the official result. The third strain, DQ354120, was classified by *st* as CRF01AE recombinant, although its official recombinant form is CRF1501B.

3.4.4. The analysis of an avian pathogenic *Escherichia coli* strain

Recently, Johnson *et al*. (2007) compared an avian pathogenic strain of *Escherichia coli* (APEC), O1:K1:H7, to strain K-12 MG1655, and 3 uropathogenic *E. coli* (UPEC) strains: CFT073, UTI89, and 536. Their result, based on chromosome alignment and BLAST (Altschul *et al*., 1990) comparison of proteins, suggests that the genome of O1:K1:H7 is most closely related to the genome of UTI89. However, they also showed that a genomic island of O1:K1:H7 (between nucleotide positions 4711722 and 4769233) exists only in strain CFT073, and not in the genomes of the other three strains, K-12 MG1655, UTI89 and 536.

In the *st* analysis of *E. coli* O1:K1:H7, I used an extended set of subject sequences to show the scalability of the method. Thus, the set of 4 subject strains (K-12 MG1655, CFT073, UTI89, and 536) was extended to set of 13 *E. coli* genomes comprising 63 Mb. The strains used as subject sequences were also used by van Passel *et al*. (2008) for the phylogenetic analysis of *E. coli* strains. The *st* analysis agreed with the previous results: most of the O1:K1:H7 genome (around 90%) is most closely related to UTI89. In addition, in the *st* analysis the segment from positions 4715950 to 4763749 was predicted to be the most closely related sequence of O1:K1:H7 is CFT073, which corresponds to the results reported by Johnson *et al*. (2007) for positions 4711722 and 4769233 (see Figure 3-18). In conclusion, the *st* results agree with the previous results showing that UPEC and APEC are more closely related to each other than to other *E. coli* strains.

The whole *st* comparison of O1:K1:H7 to 13 subject sequences took less than 6 minutes on our test computer, and the memory usage peak was 1.5 GB.

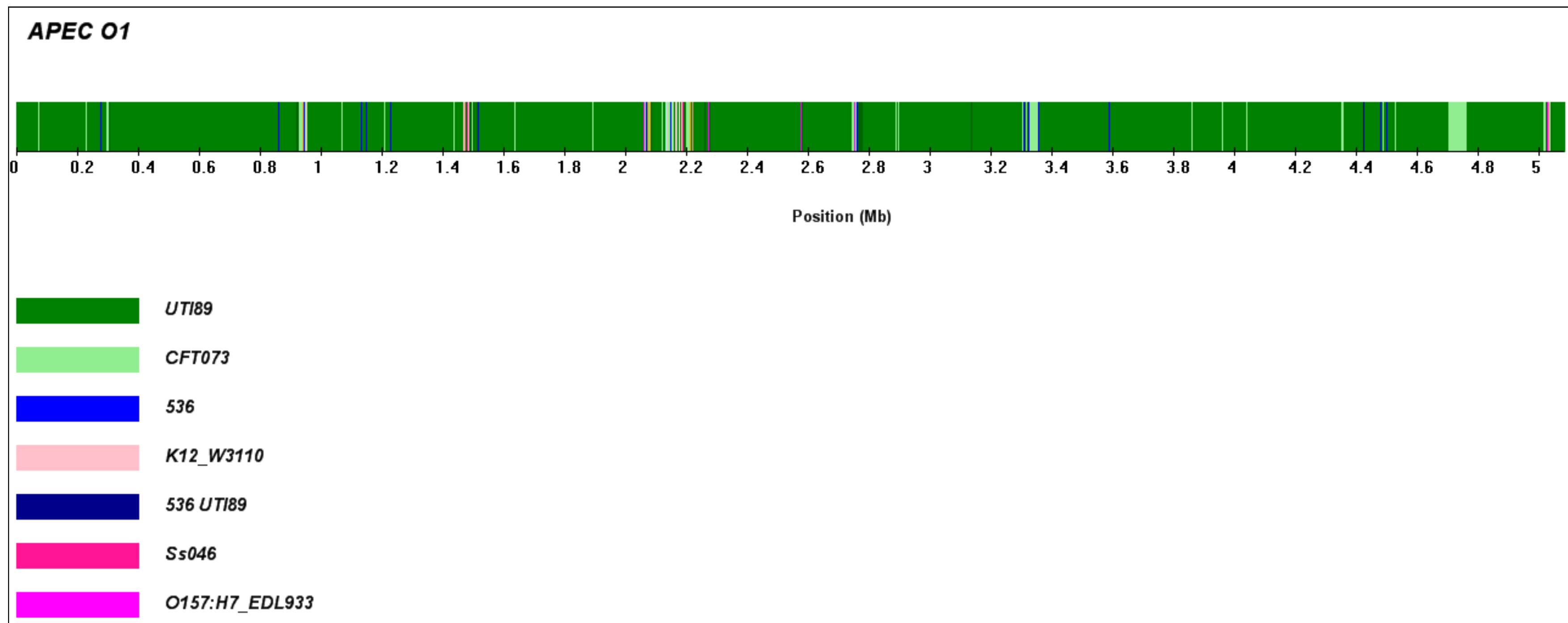


Figure 3-18. The *st* analysis of an avian pathogenic E. coli strain, O1:K1:H7. The genome of O1:K1:H7 was compared to the complete genomes of 13 *E. coli* strains, including UPEC strains (UTI89, CFT073, 536). The results show that O1:K1:H7 is the most similar to a UPEC strain UTI89 (dark green rectangles), but there is a significant segment from approximately 4.71 Mb to 4.77 Mb (light green rectangle) where O1:K1:H7 is the most similar to CFT073, which is in agreement with a previous study (Johnson *et al*., 2007).

3.5. Discussion

Recombination is typically detected from the conflicting phylogenies along genome. However, recombination detection based on the phylogeny-based methods does not scale well for large genomes; e.g., SCUEAL-based analysis of a recombinant HIV-1 strain took more than 6 hours, while the same analysis performed by *st* lasted less than a second (Section 3.4.2). To complement phylogeny-based methods on genome-scale, I developed an efficient solution for the detection of locally homologous regions, which is then applied to recombination detection (Section 3.2). In particular, the implementation of the solution, in the program *st*, is designed as a subtyping tool. That is, *st* determines the subtype or the mosaic form of a query sequence; both the recombination breakpoints, and the parental sequences.

As a part of this solution, I developed a new algorithm (Algorithm 2, Section 3.2.4), that efficiently finds intervals representing exact matches between a query sequence, and one or more subject sequences. The locally homologous regions are then derived from a sliding window analysis of the list of intervals (Section 3.2.5). For a single query and n subject (parental) sequences of length l, the time required for the computation of an interval list is $O(l(n + \log l))$. This time complexity becomes $O(nl)$ for $n >> \log l$. Hence, for the data set with numerous subject sequences, the recombinant form of a single query sequence is computed in linear time with respect to the size of the data set. In addition, the program *st* enables the computation of recombinant forms of multiple queries in a single run. In particular, determining the subtype of m queries in comparison to n subject sequences takes $O(l(n + m(1 + \log l)))$ time, which is significantly faster than computing each query on its own. This scales very well for large genomes. Moreover, to assess *st* in the context of established subtyping tools, I compared its prediction accuracy and efficiency to other subtyping tools (Section 3.4): to two popular alignment-free tools (Rozanov *et al*., 2004; Wu *et al*., 2007), and a recently developed phylogeny-based tool, SCUEAL (Kosakovsky Pond *et al*., 2009).

The *st* run-time scales well for large data sets. For example, the *st* analysis of an avian pathogenic *Escherichia coli* strain, when compared to the set of 13 *E. coli* and

Shigella strains (the total size of the data set is 65 million base pairs), lasted 6 minutes (Section 3.4.4). However, for the analysis of this strain, *st* was compared only to the alignment-based results (Johnson *et al*. 2007), since the three programs to which *st* was compared in the analysis of HIV-1 sequences (Rozanov *et al*., 2004; Wu *et al*., 2007; Kosakovsky Pond *et al*., 2009) are not easily applicable to this problem. In particular, the phylogeny based method, SCUEAL, is too slow for the analysis of such a large genome. The NCBI genotyping tool (Rozanov *et al*., 2004) is a web-based tool, with currently no stand-alone version, and the program developed by Wu *et al*. (2007) depends on HIV-specific parameters.

Next, I compared the *st* prediction accuracy to other subtyping tools (Section 3.4). In comparison to alignment-free methods (Rozanov *et al*., 2004; Wu *et al*., 2007), *st*-based results were on the same level or better than the results obtained by the other two methods (Section 3.4.3). However, a phylogeny-based tool, SCUEAL, outperformed *st* on almost all simulated data sets (Section 3.3.3). Nevertheless, in the analysis of the HIV-1 strain A_DQ083238, both programs predicted that around 30% of A_DQ083238 genome is of C subtype (Section 3.4.2; Figures 3-15, 3-16 and 3-17). The rest of the genome was mostly classified as A subtype by both *st* and SCUEAL. In addition, SCUEAL-based results also indicated the existence of smaller regions which are most closely related to an ancestral strain of A/G, and A/G/H/J subtypes. Finally, in the analysis of a large genome of an avian pathogenic *Escherichia coli* strain (Section 3.4.4; Figure 3-18) the results obtained by *st* agreed with the alignment-based analysis (Johnson *et al*., 2007).

The analysis of simulated data sets showed that *st* performs best when applied to closely related sequences (Section 3.3.2): the highest precision in determining the recombination breakpoints and parental sequences is obtained if a query and the most similar subject sequence are separated between 0.01 and 0.1 substitutions per site (Sections 3.3.2 and 3.3.3). Interestingly, the *st* prediction accuracy only slightly declines with the number of recombination breakpoints, i.e., the results of the analysis of a recombinant containing 49 breakpoints was only slightly worse than the results for the recombinant with 4 breakpoints (Section 3.3.2). Finally, these simulations provide a clue for choosing the window size in *st* windows analysis, i.e. the expected size of the recombinational fragment should determine the window size.

Specifically, the window size should not exceed the expected fragment length, for example: for the detection of fragments of length 2000 base pairs, the best results are obtained for the window size 600-2000 base pairs (Section 3.3.2; Figures 3-8 to 3-10).

4. Conclusion

Modern biology relies on the comparison of sequences, which is typically based on their alignment. However, the alignment of whole genomes is a computationally challenging task. The goal of my thesis was to find the alignment-free solutions for two problems: (i) efficient computation of K_r pair-wise distances between genomes, and (ii) detecting local sequence similarity (homology).

As a solution to the first problem, I developed a new algorithm (Algorithm 1), which computes pair-wise distances between all input sequences in a single traversal of a generalized suffix tree of all sequences (Domazet-Lošo and Haubold, 2009). I implemented the algorithm in the program *kr* version 2, and successfully applied it on both assembled and unassembled genomes (Domazet-Lošo and Haubold, 2009). The analysis of *kr* 2 showed the significant speed gain over the previous solution (*kr* 1), and efficiency and scalability in the analysis of large data sets. Thus, it is particularly suitable for the clustering of unassembled genomes, and the rapid computation of guide trees.

As a solution to the second problem, I developed a new algorithm (Algorithm 2), which computes a list of matching intervals between a query and a set of subject sequences in a single traversal of a generalized suffix tree of all input sequences. The implementation of this solution in the program *st* efficiently detects locally homologous regions, and was successfully applied to the recombination detection of HIV-1 circulating recombinant forms, and the analysis of an avian pathogenic *Escherichia coli* strain. The analysis of *st* showed that it is particularly useful for the rapid detection of longer recombinational fragments in large genomes. As a possible future direction of this research, the extension of Algorithm 2 could be used for the computation of sequence alignment: the locally similar regions found by this algorithm can represent the alignment anchors (the initial regions from which the alignment can be extended).

Finally, the applications of Algorithm 1 and Algorithm 2 do not have to be restricted to biological sequence analysis, i.e. they can be applied for the analysis of regular text. For example, Algorithm 1 can be used for clustering similar text, and

Algorithm 2 can be used for finding matching regions between a query text and a text database.

5. References

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays, *J. Disc. Alg.* 2, 53–86.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. 1990. Basic local alignment search tool. *J. Mol. Biol*. 215, 403–410.

Apostolico, A., and Denas, O. 2008. Fast algorithms for computing sequence distances by exhaustive substring composition, Algorithms Mol. Biol. 3, 13-21.

Benson, D. A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Sayers, E.W. 2009. GenBank. *Nucleic Acids Res*. 3, D26-D31.

Blaisdell, B.E. 1986. A measure of the similarity of sets of sequences not requiring sequence alignment. *Proc. Natl. Acad. Sci*. USA 83, 5155–5159.

Boni, M. F., Posada, D., Feldman, M. W. 2007. An exact nonparametric method for inferring mosaic structure in sequence triplets. *Genetics* 176, 1035-1047.

Bonnet, E., and Van de Peer,Y. 2002. Zt: a software tool for simple and partial Mantel tests. *J. Stat. Softw*. 7, 1–12.

Boyer, R. S., and Moore, J. S. 1977. A fast string-searching algorithm, *Comm.* ACM 20, 762-772.

Bray, N., and Pachter, L. 2004. MAVID: Constrained ancestral alignment of multiple sequences. *Genome Res. 14*, 693–699.

Burrows, M., and Wheeler, D. 1994. A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation

Cartwright, R.A. 2005. DNA assembly with gaps (Dawg): simulating sequence evolution. *Bioinformatics* 21 (Suppl. 3), iii31–iii38.

Do, C. B., Mahabhashyam, M. S., Brudno, M., Batzoglou, S. 2005. ProbCons: Probabilistic consistency-based multiple sequence alignment. *Genome Res*. 15, 330–340.

Domazet-Lošo, M., and Haubold, B. 2009. Efficient Estimation of Pairwise Distances between Genomes, *Bioinformatics* 25, 3221-3227.

Drosophila 12 Genomes Consortium, 2007. Evolution of genes and genomes on the *Drosophila* phylogeny. *Nature* 450, 203-218.

Dykhuizen, D. and Green, L. 1991. Recombination in Escherichia coli and the definition of biological species. *J. Bacteriol.* 173, 7257–7268.

Edgar, R. C., 2004. MUSCLE: multiple sequence alignment with high accuracy and high throughput, *Nucleic Acids Res*. 32, 1792–1797.

Edgar, R. C., and Batzoglou, S. 2006. Multiple sequence alignment. *Curr. Opin. Struct. Biol.* 16, 368–373.

Felsenstein, J. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol*. 17, 368-376.

Felsenstein, J. 1993. PHYLIP (Phylogeny Inference Package) version 3.5c. Distributed by the author. Department of Genetics, University of Washington, Seattle

Ferragina, P., and Manzini, G. 2000. Opportunistic Data Structures with Applications. *In* Proceedings of the 41st Annual Symposium on Foundations of Computer Science, Redondo Beach, CA, USA, pp. 390-398

Ferragina, P., and Manzini, G. 2005. Indexing compressed texts. *Journal of the ACM* 52, 552–581.

Ferragina, P., González, R., Navarro, G., and Venturini, R. 2008. Compressed text indexes: from theory to practice!.*ACM Journal of Experimental Algorithmics* 13.

Grossi, R., and Vitter, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *In* Proc. 32nd ACM Symposium on Theory of Computing (STOC), pp. 397-406.

Gusfield, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

Harel, D., and Tarjan, R. E. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*. 13: 338–355.

Haubold, B., and Pfaffelhuber, P. 2008. ms2dna version 1.6. Available at: http://guanine.evolbio.mpg.de/mlDiv

Haubold, B., and Wiehe, T. 2006. How repetitive are genomes? *BMC Bioinformatics* 7, 541-550.

Haubold, B., Domazet-Lošo, M., and Wiehe T. 2008. An Alignment-Free Distance Measure for Closely Related Genomes, *In* Nelson, C.E., and Vialette, S., eds., RECOMB-CG 2008, LNBI 5267, Springer-Verlag Berlin Heidelberg, pp. 87–99.

Haubold, B., Pfaffelhuber, P., Domazet-Lošo, M., and Wiehe, T. 2009. Estimating mutation distances from unaligned genomes, *J. Comput. Biol.* 16, 1487–1500.

Haubold, B., Pierstorff, N., Möller, F., and Wiehe, T. 2005. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics* 6, 123-133.

Hirschberg, D.S. 1975. A linear space algorithm for computing maximal common subsequences, *Comm.* ACM 18, 341-343.

Hoare, C.A.R. 1962. Quicksort. *Computer Journal* 5, 10-15.

Höhl, M., and Ragan, M. 2007. Is multiple-sequence alignment required for accurate inference of phylogeny? *Syst. Biol.* 56, 206–221.

Höhl, M., Kurtz, S. and Ohlebusch, E. 2002. Efficient multiple genome alignment. *Bioinformatics* 18(Suppl. 1):S312–S320.

Höhl, M., Rigoutsos, I., and Ragan, M.A. 2006. Pattern-based phylogenetic distance estimation and tree reconstruction. *Evol. Bioinform. Online* 2, 359–375.

Hudson, R. R. 2002. Generating samples under a Wright-Fisher neutral model. *Bioinformatics 18*, 337-338

Hunt, E. 2003. The Suffix Sequoia Index for Approximate String Matching. Technical Report TR-2003-135, Department of Computer Science, University of Glasgow, Glasgow.

Johnson, J. T., Kariyawasam, S., Wannemuehler, Y., Mangiamele, P., Johnson, S. J., Doetkott, C., Skyberg, J. A., Lynne, A. M., Johnson, J. R., and Nolan, L. K. 2007. The genome sequence of avian pathogenic *Escherichia coli* strain O1:K1:H7 shares strong similarities with human extraintestinal pathogenic *E. coli* genomes. *J. Bacteriol*. 189, 3228-3236.

Jukes, T. H., and Cantor, C.R. 1969. Evolution of protein molecules. *In* Munro, H.N., ed., *Mammalian Protein Metabolism*, volume 3. Academic Press, New York, 21–132.

Kärkkäinen, J., and Sanders P. 2003. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, Springer, pp. 943-955.

Karlin, S., and Altschul, S. F. 1990. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl. Acad. Sci*. 87, 2264-2268.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. *In* Proc. Annual Symposium on Combinatorial Pattern Matching, LNCS 2089, Springer-Verlag, Berlin, pp. 181–192.

Katoh, K., Hiroyuki, T. 2008. Recent developments in the MAFFT multiple sequence alignment program, *Brief. Bioinform.* 9, 286-298

Keeling, P. J., and Palmer, J. D. 2008. Horizontal gene transfer in eukaryotic evolution, *Nat. Rev. Genet*. 9, 605–618.

Kim, D. K., Sim, J. S., Park, H., and Park, K. 2005. Constructing suffix arrays in linear time. *J. Discrete Algorithms* 3, 126–142.

Kimura, M. 1980. A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *J. Mol. Evol.* 16, 111-120.

Knuth, D.E., Morris J.H., and Pratt, V.R. 1977. Fast pattern matching in strings. *SIAM J. Computing* 6, 323-350.

Ko, P., and Aluru, S. 2003. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching (CPM 03)*, Springer, 203-210.

Kosakovsky Pond, S. L., Posada, D., Stawiski, E., Chappey, C., Poon, A. F. Y., Hughes, G., Fearnhill, E., Gravenor, M. B., Leigh Brown, A. J., and Frost S. D. W. 2009. An Evolutionary Model-Based Algorithm for Accurate Phylogenetic Breakpoint Mapping and Subtype Prediction in HIV-1. *PLoS Comput Biol* 5 (11), e1000581.

Kumar, S., Nei, M., Dudley, J., and Tamura, K. 2008. MEGA: a biologist-centric software for evolutionary analysis of DNA and protein sequences. *Brief Bioinform.* 9, 299-306.

Kurtz, S. 1999. Reducing the space requirements of suffix trees. *Software: Practice and Experience* 29, 1149–1171.

Kurtz, S., Choudhuri, J.V., Ohlebusch, E., Schleiermacher, C., Stoye, J. and Giegerich, R. 2001. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res*., 29, 4633–4642.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*. 10, R25

Larkin, M.A., Blackshields, G., Brown, N.P., Chenna, R., McGettigan, P.A., McWilliam, H., Valentin, F., Wallace, I.M., Wilm, A., Lopez, R., Thompson, J.D., Gibson, T.J., and Higgins, D.G. 2007. ClustalW and ClustalX version 2. *Bioinformatics* 23, 2947-2948.

Leitner, T., Korber, B., Daniels, M., Calef, C., and Foley B. 2005. HIV-1 Subtype and Circulating Recombinant Form (CRF) Reference Sequences.

Li, H., and Durbin, R. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 1754-1760.

Li, H., and Durbin, R. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 589–595.

Li, M., Badger, J. H., Chen X., Kwong, S., Kearney, P., Zhang, H. 2001. An information based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics* 17, 149-154.

Lipman, D. J. and Pearson, W. R. 1985. Rapid and sensitive protein similarity searches. *Science* 227, 1435-1441.

Manber, U., and Myers, E.W. 1993. Suffix arrays: a new method for on-line string searches, *SIAM J. Comput.* 22, 935–948.

Maniscalco, M. A., and Puglisi, S. J. 2006. Faster lightweight suffix array construction. *In* Ryan, J., and Dafik, eds., *Proc*. of *17th Australasian Workshop on Combinatorial Algorithms*, Univ. Ballavat, Ballavat, Victoria, Australia, pp. 16-29.

Maniscalco, M. A., and Puglisi, S. J. 2007. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics 12*, 1-31.

Mantel, N. 1967. The detection of disease clustering and a generalized regression

approach. *Cancer Res.* 27, 209-220.

Manzini, G. 2004. Two space-saving tricks for linear-time LCP computation. *In* Hagerup, T., and Katajainen, J., eds., *Proc. SWAT 2004. Lecture Notes in Comput. Sci. 3111*, Springer-Verlag, Berlin, pp. 372-383.

Manzini, G. and Ferragina, P. 2004. Engineering a lightweight suffix array construction algorithm. Algorithmica 40, 33–50.

McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM.* 23, 262-272.

Michener, C. D., and Sokal, R. R. 1957. A quantitative approach to a problem in classification. *Evolution* 11, 130–162.
Navarro, G. 2004. Indexing text using the Ziv-Lempel trie. *JDA* 2, 87–114.

Navarro, G., and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys* 39, article 2.

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. *J. Mol. Biol.* 48, 443-453.

Notredame, C. 2007. Recent evolutions of multiple sequence alignment algorithms. *PLoS Comput. Biol.* 3: e123.

Notredame, C., Higgins, D., and Heringa J. 2000. T-Coffee: A novel method for multiple sequence alignments. *J. Mol. Biol.* 302, 205-217.

Otu, H. H., and Sayood, K. 2003. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics* 19, 2122–2130.

Posada, D., Crandall K. A., 2001. Evaluation of methods for detecting recombination from DNA sequences: Computer simulations. *Proc. Natl. Acad. Sci.* USA 98, 13757-13762.

Posada, D., Crandall, K. A., and Holmes, E.C. 2002. Recombination in evolutionary genomics. *Annu Rev Genet*. 36, 75–97.

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. 2007. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.* 39, 1-31.

Ristov, S. 2003. A Note on Indexing DNA and Protein Sequences. *In* Bohanec, M., Filipič, B., and Gams, M., eds., *Proc.* 6th Intl. Multi-Conference Information Society IS 2003, Vol A, Intelligent and Computer Systems, Ljubljana, Institut "Jožef Štefan", pp. 121-126.

Robinson, D. F., and Foulds, L. R. 1981. Comparison of phylogenetic trees. *Math. Biosci*. 53, 131-147.

Rozanov, M., Plikat, U., Chappey, C., Kochergin, A., and Tatusova, T. 2004. A web-based genotyping resource for viral sequences. *Nucleic Acids Res*. 32, 654-659.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms* 48, 294-313.

Saitou, N., and Nei, M. 1987. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.* 4, 406-425.

Schieber, B., and Vishkin, U. 1988. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Computing* 17, 1253–1262.

Seward, J. 2007. bzip2 1.0.5, http://www.bzip.org/

Sims, G.E., Jun, S.-R., Wu, G. A., and Kim S.-H. 2009. Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions, *Proc. Natl. Acad. Sci. USA* 106, 2677-2682.

Smith, T.F., and Waterman, M.S. 1981. Identification of common molecular subsequences. *J. Mol. Biol*. 147, 195-197.

Stuart, G.W., and Berry, M.W. 2003. A comprehensive whole genome bacterial phylogeny using correlated peptide motive defined in a high dimensional vector

space. *J. Bioinf. Comp. Biol.* 1, 475–493.

Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica* 14, 249-260.

Ulitsky, I., Burstein, D., Tuller, T., Chor, B. 2006. The average common substring approach to phylogenomic reconstruction. *J. Comput. Biol.* 13, 336-350.

Välimäki, N., Gerlach, W., Dixit, K., and Mäkinen, V. 2007. Compressed suffix tree - a basis for genome-scale sequence analysis. *Bioinformatics* 23, 629-630.

van Passel, M.W., Marri, P.R., and Ochman, H. 2008. The emergence and fate of horizontally acquired genes in *Escherichia coli*. *PLoS Comput Biol* 4 (4), e1000059.

Vinga, S., and Almeida, J. 2003. Alignment-free sequence comparison—a review. *Bioinformatics* 19, 513–523.

Wang, L. and Jiang, T. 1994. On the complexity of multiple sequence alignment. *J. Comput*. *Biol*. 1, 337–348.

Weiner, P. 1973. Linear pattern matching algorithm. *In* Proc. *14th Annual IEEE Symposium on Switching and Automata Theory*, Washington DC, pp. 1-11.

Westesson, O., Holmes, I. 2009. Accurate Detection of Recombinant Breakpoints in Whole-Genome Alignments. PLoS Comput Biol 5: e1000318.

White, M. A., Ané, C., Dewey, C. N, Larget, B. R., and Payseur B. A. 2009. Fine-Scale Phylogenetic Discordance across the House Mouse Genome. PLoS *Genet* 5 (11): e1000729.

Williams, J.W.J. 1964. Algorithm 232 – Heapsort. *Comm*. *ACM* 7, 347-348.

Wu X., Cai, Z., Wan, X., Hoang, T., Goebel, R., and Lin, G. 2007. Nucleotide composition string selection in HIV-1 subtyping using whole genomes. *Bioinformatics* 23, 1744–1752.

Ziv, J., and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. IEEE *Trans*. *Inform*. *Theory* IT-24, 530-536.

6. Electronic Sources

1000 Genomes – A Deep Catalog of Human Genetic Variation. May, 2010. Web. June 10, 2010.

<http://www.1000genomes.org>

1000 Genomes Project. June 4, 2010. Web. June 10, 2010.

<http://en.wikipedia.org/wiki/1000_Genomes_Project>

GenBank. January 29, 2010. Web. June 9, 2010.

<www.ncbi.nlm.nih.gov/genbank>

Gregory T.R. 2005. Animal Genome Size Database. n. d. Web. June 12, 2010.

<http://www.genomesize.com>

HIV sequence database. January 26, 2010. Web. June 10, 2010.

<http://www.hiv.lanl.gov>

The 1KP Project. n. d. Web. June 4, 2010

<http://www.onekp.com>

7. List of abbreviations and symbols

Symbol	Explanation
Σ	alphabet
T	text – a longer string (Chapter 1); a (generalized) suffix tree (Chapters 2 and 3)
P	pattern – a shorter string
z	number of occurrences of a pattern (P) in a text (T)
S	arbitrary string (Chapter 1)
$\|S\|$	length of a string S
$S[i .. j]$	a substring of a string S starting at the position i, and ending at the position j
$	sentinel character of a string S
$H_k(S)$	k-th order entropy of S
BWT	Burrows-Wheeler Transform
SA	suffix array
ST	suffix tree
c	a character from Σ
$C(c)$	function that returns number of characters smaller than c in a string S
$Occ(c, i)$	number of occurrences of c in B[1..i], where B is a *BWT* of a string S
$[L_P, R_P]$	Interval of *SA* which covers occurrences of a prefix P in a string S
con	k-length substring (context) of a string S
S^{con}	concatenation of characters following *con* in a string S, taken from the left to the right of S
n	number of sequences (Chapter 2); number of subject sequences (Chapter 3)
m	number of query sequences
l	length of a sequence
$\mathbf{S} = \{S_1, .., S_n\}$	a set of nucleotide sequences (Chapter 2); a set of subject sequences (Chapter 3)

$\boldsymbol{Q} = \{Q_1, .., Q_m\}$	a set of query sequences
(S_i, S_j)	a pair of sequences from $\{S_1, .., S_n\}$
$h_{i,j,p}$	a shortest prefix of $S_i[p .. \lvert S_i \rvert]$ absent from S_j (*shustring*)
$o_{i,j}$	observed average shustring length for a pair (S_i, S_j)
$d_{i,j}$	number of pair-wise mismatches per nucleotide between S_i and S_j
$H_{i,p}$	the maximal value of $h_{i,j,p}$ for a position p in a query Q_i when compared to subjects $S_j \in \boldsymbol{S}$, $j = 1, .., n$, and $i \neq j$
K	Jukes-Cantor evolutionary distance
K_r	evolutionary distance measure between S_i and S_j, where $d_{i,j}$ is converted to the number of nucleotide substitutions using Jukes-Cantor formula (Chapter 2)
$G_i = G_{i,1}..G_{i,k}$	a mosaic structure of Q_i, where each segment $G_{i,d}$ is the most similar to members of $\boldsymbol{S_{i,d}}$, where $\boldsymbol{S_{i,d}}$ is a subject of $\boldsymbol{S}$
$\boldsymbol{S_{i,d}}$	a subset of subjects of $\boldsymbol{S}$ to which Q_i is the most similar over $G_{i,d}$
$\boldsymbol{S_{i,p}}$	a subset of subjects of $\boldsymbol{S}$ to which Q_i is the most similar at a position p
$\pi_\mathrm{A}, \pi_\mathrm{C}, \pi_\mathrm{G}, \pi_\mathrm{T}$	stationary nucleotide frequencies
$\theta_\mathrm{AC}, \theta_\mathrm{AG}, \theta_\mathrm{AT}, \theta_\mathrm{GC}, \theta_\mathrm{CT}, \theta_\mathrm{GT}$	substitution rates
w	length of a sliding window in a sliding window analysis (in the program *st*)
f	minimal length of a recombinant fragment to be considered as a relevant (in the program *st*)
kb	kilo base pair (thousand base pairs)
Mb	mega base pair (million base pairs)
Gb	giga base pair (billion base pairs)

Abstract

Sequence comparison is an essential tool in modern biology. It is used to identify homologous regions between sequences, and to detect evolutionary relationships between organisms. Sequence comparison is usually based on alignments. However, aligning whole genomes is computationally difficult. As an alternative approach, alignment-free sequence comparison can be used. In my thesis, I concentrate on two problems that can be solved without alignment: (i) estimation of substitution rates between nucleotide sequences, and (ii) detection of local sequence homology. In the first part of my thesis, I developed and implemented a new algorithm for the efficient alignment-free computation of the number of nucleotide substitutions per site, and applied it to the analysis of large data sets of complete genomes. In the second part of my thesis, I developed and implemented a new algorithm for detecting matching regions between nucleotide sequences. I applied this solution to the classification of circulating recombinant forms of HIV, and to the analysis of bacterial genomes subject to horizontal gene transfer.

Keywords: alignment-free method, evolutionary distance, local sequence homology, genome comparison, HIV, horizontal gene transfer, suffix tree, suffix array, shortest unique substringing.

Sažetak

Algoritmi za učinkovitu usporedbu sekvenci bez korištenja sravnjivanja

Uspoređivanje sekvenci je osnovni alat u modernoj biologiji, a koristi se za pronalaženje homolognih dijelova između sekvenci te za otkrivanje evolucijskih odnosa između organizama. Uspoređivanje je sekvenci obično temeljeno na sravnjivanju. Međutim, sravnjivanje cijelih genoma je računalno zahtijevan postupak. Kao alternativni pristup, mogu se koristiti metode koje ne koriste sravnjivanje sekvenci. U sklopu svoje doktorske disertacije, koristila sam pristup koji ne zahtijeva sravnjivanje sekvenci u rješavanju dvaju problema: (i) procjena brzine supstitucije između nukleotidnih sekvenci; (ii) određivanje lokalne homologije između nukleotidnih sekvenci. U sklopu prvog dijela disertacije razvila sam i implementirala algoritam za učinkovito računanje procjene relativnog broja supstitucija između dviju nukleotidnih sekvenci bez korištenja sravnjivanja, koji sam primijenila za analizu velikih skupova cijelih genoma. U drugom dijelu disertacije razvila sam i implementirala novi algoritam za određivanje jednakih dijelova između nukleotidnih sekvenci. Rješenje sam primijenila za određivanje roditeljskih tipova rekombinantnih oblika virusa HIV te za analizu bakterijskih genoma pod utjecajem horizontalnog prijenosa gena.

Ključne riječi: usporedbe sekvenci bez sravnjivanja, lokalna homologija, evolucijska udaljenost, usporedba genoma, HIV, horizontalni prijenos gena, sufiksno stablo, sufiksno polje, najkraći jedinstveni podniz.

Curriculum vitae

Mirjana Domazet-Lošo was born in 1976 in Zagreb, Croatia. She studied computer science at the Faculty of Electrical Engineering and Computing, University of Zagreb, where she got her Diploma in 1999, and Master's degree in 2006. She began her doctoral studies at the Faculty of Electrical Engineering and Computing, University of Zagreb in the Fall of 2006. She was a guest doctoral student in the Bioinformatics group, Department of Evolutionary Genetics, Max-Planck-Institute for Evolutionary Biology, Plön, Germany. She has been a researcher and a teaching assistant at the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb since 1999. She has contributed to publications in international peer-reviewed journals and to international conferences.

Publications

- Domazet-Lošo, M., and Haubold, B. 2009. Efficient Estimation of Pairwise Distances between Genomes, *Bioinformatics* 25, 3221-3227.

- Haubold, B., Pfaffelhuber, P., Domazet-Lošo, M., and Wiehe, T. 2009. Estimating mutation distances from unaligned genomes, *J. Comput. Biol.* 16, 1487–1500.

- Haubold, B., Domazet-Lošo, M., and Wiehe T. 2008. An Alignment-Free Distance Measure for Closely Related Genomes, *In* Nelson, C.E., and Vialette, S., *eds*., RECOMB-CG 2008, LNBI 5267, Springer-Verlag Berlin Heidelberg, pp. 87–99.

- Baranović, M., Madunić, M., Mekterović, I. 2003. Data Warehouse as a Part of the Higher Education Information System in Croatia. *In* Budin, L., Lužar-Stiffler, V.; Bekić, Z. and Hljuz Dobrić, V. *eds*., Proc. of the 25th International Conference on Information Technology Interfaces, SRCE, Zagreb, pp. 121-126.

Životopis

Mirjana Domazet-Lošo je rođena 1976. godine u Zagrebu. Studirala je računarstvo na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, gdje je diplomirala 1999. te magistrirala 2006. godine. Doktorski studij računarstva upisala je u jesen 2006. godine na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Bila je gost-doktorski student u Bioinformatičkoj grupi Odjela za evolucijsku genetiku Instituta Max Planck za evolucijsku biologiju u Plönu (Njemačka). Zaposlena je kao znanstveni novak na Zavodu za primijenjeno računarstvo Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu od 1999. godine. Objavila je radove u časopisima s međunarodnom recenzijom i na međunarodnim znanstvenim skupovima.

Objavljeni znanstveni radovi

- Domazet-Lošo, M., Haubold, B. 2009. Efficient Estimation of Pairwise Distances between Genomes, *Bioinformatics* 25, 3221-3227.

- Haubold, B., Pfaffelhuber, P., Domazet-Lošo, M., Wiehe, T. 2009. Estimating mutation distances from unaligned genomes, *J. Comput. Biol.* 16, 1487–1500.

- Haubold, B., Domazet-Lošo, M., Wiehe T. 2008. An Alignment-Free Distance Measure for Closely Related Genomes, *In* Nelson, C.E., and Vialette, S., *eds*., RECOMB-CG 2008, LNBI 5267, Springer-Verlag Berlin Heidelberg, pp. 87–99.

- Baranović, M., Madunić, M., Mekterović, I. 2003. Data Warehouse as a Part of the Higher Education Information System in Croatia. *In* Budin, L., Lužar-Stiffler, V.; Bekić, Z. and Hljuz Dobrić, V. *eds*., Proc. of the 25th International Conference on Information Technology Interfaces, SRCE, Zagreb, pp. 121-126.