UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
NEURO- UND BIOINFORMATIK

Fast Multiple Sequence Alignment

Schnelles Multiples Sequenz Alignment

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
David Gmelin

ausgegeben und betreut von
Prof. Dr. Bernhard Haubold

Lübeck, den 22. September 2022 (modified November 10, 2022)

IM FOCUS DAS LEBEN

Notice

For this version, the '*Eidesstattliche Erklärung*' was removed for privacy reasons.
I also adapted a twisted number in the covid results table.

David Gmelin - Lübeck, den November 10, 2022

Acknowledgements I would like to thank two people in particular. There is Bernhard, for the opportunity to write my thesis in his group and for the expertise. Also Swantje, for the emotional support and for understanding my occasional moods.

Abstract Sequencing technologies are continuously improving and provide access to an increasing amount of data. This gives rise to many opportunities to gather new insights about the sequenced organisms. However, handling such volumes of data is a challenge on its own and established, alignment-based, methods for sequence comparison are reaching their capacities. The alternative, *alignment-free* methods scale better to large datasets and are especially useful for phylogeny reconstruction. But their effectiveness comes with the disadvantage of loosing information about the underlying alignment structure (Vinga, 2014). In this thesis, the approach of *anchor alignments* is implemented. Anchor alignments have already shown good results in *phylonium* (Klötzl and Haubold, 2019) for *alignment-free* distance estimation. The goal of this thesis is to make the underlying alignment accessible and to evaluate the results against alignment-based methods.

The resulting program *par* is faster than classical alignment-based approaches. The alignments are accurate on very closely related genomes that are currently collected during pangenomic outbreaks. However, as the sequences become more divergent, the accuracy starts to drop quickly.

Kurzfassung Methoden zur Genomsequenzierung verbessern sich stetig und ermöglichen den Zugang zu immer größeren Datenmengen. Dadurch ergeben sich viele Möglichkeiten, neue Erkenntnisse über die sequenzierten Organismen zu gewinnen. Die Bewältigung solcher Datenmengen ist jedoch eine Herausforderung an sich bei der die etablierten, auf Alignment basierenden Methoden zum Sequenzvergleich an ihre Grenzen stoßen. Die Alternative, *alignment-freie* Methoden eignen sich besser für große Datenmengen und sind insbesondere für die Rekonstruktion von Phylogenien nützlich. Ihre Effektivität hat den jedoch den Nachteil, dass die zugrundeliegenden Alignments verloren gehen. In dieser arbeit ist der Ansatz der *Anker-Alignments* implementiert. Mit Anker-Alignments konnten bereits in *phylonium* (Klötzl and Haubold, 2019) als *alignment-freier* Ansatz zur Distanzschätzung gute Ergebnisse erziehlt werden. Das Ziel dieser Arbeit ist es, das zugrundeliegende Alignment zugänglich zu machen und die Ergebnisse mit Alignment-basierten Methoden zu vergleichen.

Das daraus resultierende Programm *par* ist schneller als klassische, alignment-basierte Ansätze. Die Alignments sind präzise für sehr eng verwandte Genome, die derzeit bei pangenomischen Ausbrüchen gesammelt werden. Bei weiter entfernten Sequenzen nimmt die Genauigkeit jedoch schnell ab.

Contents

1 Introduction **1**

2 Theoretical Background **3**
2.1 Sequence Alignment . . . 3
2.2 Alignment-Free Distance Estimation . . . 6
2.3 String Matching . . . 7
2.4 (Enhanced) Suffix Arrays . . . 9
2.5 Suffix Array Construction . . . 12

3 Implementation **15**
3.1 Input and Preprocessing . . . 15
3.2 ESA revisited . . . 16
3.3 Finding the Anchors . . . 17
3.4 Sorting and optional filtering . . . 18
3.5 Pile Anchors on Reference . . . 19

4 Evaluation Method **25**
4.1 Scoring Approach . . . 26
4.2 Data . . . 26
4.2.1 Simulated data . . . 27
4.2.2 Real data . . . 27

5 Results **29**
5.1 Benchmarking Suffix Array Libraries . . . 29
5.2 Simulated data . . . 29
5.3 Real Data . . . 32
5.4 Comparison to Phylonium . . . 35

6 Discussion **37**

1 Introduction

A good way to compare items is to place them next to each other and examine their similarities. On the most abstract level, the alignment of genomes in biology can be described as such a form of comparison. Sequence alignment is a very common early step in molecular biology and, because of its complexity, it is a cornerstone of the link between biology and computer science, bioinformatics. In a survey published in *Nature* that ranks the top 100 most cited papers of all time (Van Noorden et al., 2014), the popular alignment tools ClustalW (Thompson et al., 1994) and BLAST (Altschul et al., 1990) came 10th and 12th, respectively.

Sequence alignments are useful for a wide range of applications in biology. They can be used for phylogenetic reconstruction, to detect patterns within genomes, for comparative genomics, and for predicting protein function (Iantorno et al., 2014; Zhang et al., 2022). With the ongoing advances in sequence technologies, an increasing number of genome sequences are becoming available. This generates the interest in alignment strategies that help analysing genomes. However, as even more and longer sequences become available, the runtime and memory consumption of these methods may become unacceptable or even unfeasible. Tools that are able to handle larger genomes require large server infrastructure and take several hours to complete (Earl et al., 2014). This sets the stage for alignment-free methods. Alignment-free methods are based on alternative dissimilarity metrics that allow to skip the expensive process of residue-by-residue sequence alignment (Vinga and Almeida, 2003). Alignment-free methods have been increasingly popular in the last two decades, especially in phylogeny reconstruction. More recently, alignment-free methods have become able to not only estimate phylogenetic tree topologies but also to determine phylogenetic distances correctly (Morgenstern, 2021). In Zielezinski et al. (2017) the results of various alignment-free tools were compared in detail.

In the alignment-free tools *andi* (Haubold et al., 2014) and its faster successor *phylonium* (Klötzl and Haubold, 2019) the authors implemented the idea of *anchor distances* to estimate evolutionary distances. Anchor distances are based on micro-alignments of regions that are anchored by exact matches. Both approaches proved to be orders of magnitudes faster than classical alignment tools while still being accurate on closely related genomes. Even among similar alignment-free methods they were found to be among the fastest (Zielezinski, 2019). However, the actual sequence alignments that are used in *andi* and *phylonium* to generate the distance matrices stay implicit and cannot be accessed. The goal of this thesis is to make the alignments accessible and to provide them in a way that makes them comparable to classical alignment tools. This could prove to be another advantage of the anchor matching method. The resulting explicit alignments can be evaluated in further analyses, e.g. for specific sequence segments. Additionally, alignments

allow bootstrapping of phylogenies. Bootstrapping is a common method to measure confidence of individual nodes in phylogenetic trees but require sequence alignments. As the name suggests, alignments are usually not returned in alignment-free methods.

This thesis is structured as follows: Chapter 2 surveys the theoretical background of this work. It describes different approaches on sequence comparison. Furthermore, I take a look at suffix trees and suffix arrays that are a central data structure of this thesis and in sequence comparison. In Chapter 3, I describe the details of the implementation and application of my program *par* that generates the alignments. The short Chapter 4 explains my approach to evaluate the alignments and Chapter 5 presents the results of my evaluation experiments. In Chapter 6, I will discuss the results and propose ideas for further improving *par*.

2 Theoretical Background

In this chapter I introduce the concepts underlying alignment-based sequence comparison and alignment-free sequence comparison. The field of multiple sequence alignment has been reviewed by Dewey (2019); Chatzou (2016) and Armstrong et al. (2019). Katoh (2021) provides an up-to-date collection of multiple alignment tools for specific problems. Haubold (2013) and Zielezinski et al. (2017) do the same for alignment-free sequence comparison. Following the concepts for sequence comparison, I present methods for string matching.

2.1 Sequence Alignment

The importance of sequence comparison and hence sequence alignment is described in Gusfield (1997) as the *'first fact of biological sequence analysis'*: According to Gusfield 'In biomolecular sequences (DNA, RNA or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity'.

To find regions of high similarity among sequences, genes or genomes is the goal of sequence alignment. More precisely, sequence alignment aims to find positions in two or more sequences that are homologous and assigns them to one another. In biology, the term *homology* indicates similarity in a sense that two or more genes or segments on the DNA have descended from a common ancestor due to speciation events (Kehr et al., 2014; Haubold and Wiehe, 2004).

A sequence alignment can be described as a matrix where each row belongs to a sequence to be aligned and each column corresponds to the characters of that sequence. The characters should be arranged in such a way that the number of matching (equal) characters in a column is maximal. This is done by introducing gaps ('-') at certain positions that shift the sequences by one position. An example for a pairwise alignment of two short strings can be seen in Figure 2.1.

For alignments, a distinction is made between local and global alignment. The function of local alignments is to find subregions of high similarity among the compared sequences. In this case, aligning complete sequences is not necessary or even not advised, since only parts of the sequences might be related. Global alignment, on the contrary, attempts to align the entire sequences. The optimal global alignment between a pair of sequences can be computed using the *Needleman-Wunsch*-Algorithm (Needleman and Wunsch, 1970). An algorithm to find optimal local alignments is the algorithm by Smith and Waterman (1981). In this context, optimal means that these algorithms are guaranteed to find the

-	A	G	T	A	-	C	C	A
T	A	G	A	A	A	C	C	A

Figure 2.1: Exemplary pairwise alignment for the sequences S_1=AGTACCA and S_2=TAGAAACCA. Note that the solution is not always unique, since the gap can also be placed before the second A in S_1 and not behind it.

best possible solution (Haubold and Wiehe, 2006). Multiple sequence alignment (MSA) is an extension of the pairwise alignment to more than two sequences. But multiple alignment is more than a 'generalization for generalization's sake' (Gusfield, 1997). Application areas for multiple and pairwise sequence alignments can be even inverse to each other. Whereas pairwise alignments are of great use to discover similar sequences that are not known to be similar the goal of multiple alignments is to discover unknown subpatterns and similarities for sequences that are known to be similar and related. (Gusfield, 1997)

To find the optimal alignment between multiple sequences is known to be a NP-complete problem (Wang and Jiang, 1994), which makes it not solvable for more than a few and short sequences. For this reason algorithms for MSAs are based on heuristics. Such heuristic algorithms are not guaranteed to yield optimal alignments, but they are applicable for longer sequences. The greedy *progressive multiple alignment* (Feng and Doolittle, 1987) is a standard approach in traditional multiple sequence alignment methods. Progressive alignment strategies are used in popular tools like ClustalW (Thompson et al., 1994) and MAFFT (Katoh et al., 2002) that aim to globally align the given sequences. The idea of the progressive method is to align very similar sequences first because they potentially provide more reliable information about the real alignment (Feng and Doolittle, 1987; Jones and Pevzner, 2004). For the progressive alignment the evolutionary relationship between the sequences is assumed to be known and given in form of a phylogenetic tree, the '*guide tree*'. The leafs of said tree represent sequences to be aligned. Starting at the leafs, the closest related sequences are aligned first into a '*profile*'. This profile, which is an alignment of at least two sequences, is kept and the next nearest sequence or profile is aligned. This way, the multiple alignment is build successively along the tree for each node. The final, complete alignment of all sequences is build at the root of the tree. (Katoh, 2021; Batzoglou, 2005; Chatzou, 2016). In its original form this algorithm is prone to local minima leading to suboptimal results. This means that the alignment of two very closely related sequences might be incompatible with the best possible global result. To overcome this problem, several heuristics exist to prevent progressive aligners to either avoid local minima directly or to correct errors afterwards by iteratively realigning critical parts (Zhang et al., 2022; Katoh et al., 2002). Another, even more critical limitation for these types of alignments is that they consider insertions, deletions and substitutions to be the only allowed evolutionary operations. They result in global alignments of the complete sequences from start to their end. Other, more elaborate events such as duplications and rearrangements are not considered (Dewey, 2019).

Whole genome alignment

The availability of long, complete genome sequences motivates the interest in being able to align them in full. Classical alignment algorithms are too computationally expensive and not suitable for these tasks. Additionally, it is not guaranteed that the homologous regions are collinear, i.e. are conserved in the same order (Chatzou, 2016). The reason for this might be mutation events like duplication or rearrangements but also that the genomes are not completely available, contain sequencing gaps or are in the form of 'draft' genomes (Angiuoli and Salzberg, 2010). So-called *whole genome alignment* approaches consider a multiple genome alignment not to consist of global collinear sequences but as a set of multiple blocks of segments that share high local similarity. The blocks do not have to be related among each other (Chatzou, 2016). They represent collinear, homologous and rearrangement-free regions that are often referred to as *Locally Collinear Blocks* (LCBs) (Darling et al., 2004). LCBs are much smaller than whole-genomes and can be handled by standard MSA approaches that are based on progressive alignment (Armstrong et al., 2019).

The idea behind most approaches for WGA is to distinguish between the problem of finding homologous regions and to produce an alignment on the nucleotide-level (Dewey, 2019). In a first step they search for small segments, so-called *anchors*, using fast local alignment algorithms. Next, the anchors are used to compute *chains* that form collinear and rearrangement free regions, the LCBs (Armstrong et al., 2019; Ohlebusch, 2013). The identification of LCBs is often guided using graphs, since graphs provide a convenient data structure to identify subpatterns within the alignments, e.g. with circles. Kehr et al. (2014) provides an overview over different graph structure approaches for whole genome alignment. Finding anchors and chains of collinear regions serves to reduce the search space for the sequence alignment. In a third step, regions that are much shorter than the initial sequences can be aligned using any global alignment algorithm.

Usage of Sequence Alignments

Sequence alignments play an important role in the analysis of genomic sequences. As functional elements are biologically important, they are assumed to be more restricted to evolutionary changes and remain conserved within different genomes (Batzoglou, 2005). Thus, conserved regions may indicate some type of functionality.

An important use case for sequence alignments is searching in databases. For newly sequenced genes, functions and relationships might be unknown. It is common practice to search databases for similar sequences or genes for which the function is already known (Jones and Pevzner, 2004; Zhang et al., 2022). A prominent example of such tools are tools based on the BLAST (Altschul et al., 1990). BLAST stands for *Basic Local Alignment Search Tool* and is an effective heuristic to find local alignments between DNA sequences, RNA sequences or protein sequences. It is a collection of programs that are specialized for different applications like protein-protein comparison, nucleotide-protein comparison or comparison of a large number of sequences. BLAST, also other database searches, aim

at finding regions of high local similarity (Gusfield, 1997). This helps to discover similar sequences or segments of sequences.

In addition to database searching, a comparison of sequences can reveal unknown related subpatterns. Sequence alignment can be used to find relevant regions that indicate some kind of functionality in the first place. As mentioned above, the extension from two to multiple sequences can help to discover even more or broader similarities among the sequences compared.

Reconstructing phylogenies is another important application for multiple sequence alignments. Phylogenies are a central tool for genomic research. For instance, they help, in identifying viruses during outbreaks (Gorbalenya et al., 2020). They correspond to the very intuitive metaphor of representing relationships as a tree. A famous representative of this is a sketch of the *'Tree of Life'* in one of Darwin's notebooks. Most methods for the reconstruction of phylogenies are based on sequence alignments (Felsenstein, 2004), which is still considered the gold standard. However, with the often-mentioned further increase in available genomic data, the applicability of fast whole-genome alignment methods is also reaching its limits. In the supplementary material of (Earl et al., 2014) computational resources and runtimes were published. Here it can be seen that many tools required large computer cluster and long runtimes, even for relatively few genomes. Alignments for thousands of genomes is a challenge that alignment tools cannot solve at the moment (Armstrong et al., 2019).

Besides, some alignment tools themselves rely on phylogenies as input data to guide the alignments. This seems to be a *chicken or egg dilemma*, since determining phylogenies is one of their fields of usage. For this reason, alignment-free sequence analysis offers an attractive alternative to alignment-based methods. Alignment-free methods benefit from the fact that a sequence alignment is not necessarily required to calculate the similarity between sequences (Haubold, 2013).

2.2 Alignment-Free Distance Estimation

Alignment-Free (AF) methods aim to compare sequences without actually having to align them. AF-methods have been increasingly popular for phylogeny reconstruction in the last 20 years (Zielezinski, 2019). They provide fast algorithms for distance estimation between genomes. The resulting distances can then be used to construct phylogenetic trees using standard clustering algorithms like *neighbor joining* (Saitou and Nei, 1987).
As in alignment-based methods, the idea of AF-methods is based on the assumption that similar sequences share similar subsequences (Zielezinski et al., 2017). Following Haubold (2013), AF methods can be broadly classified into methods based on word counts or on match lengths. One of the earliest methods developed is based on the comparison of common substrings of fixed length. In its initial approach, the sequence is split into words of fixed length k, the *k-mers*. Subsequently, the frequency of said k-mers is compared with other sequences. This results in an estimated distance between two sequences or in a matrix of distances for multiple sequences compared.

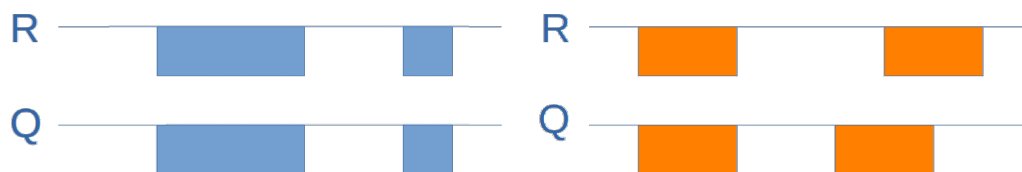


Figure 2.2: Example of anchor pairing: The blue anchors on the left are equidistant and form a pair, the orange anchors on the right are not equally spaced and are not considered. The anchors do not have to start at the same positions in the sequence.

An example for methods based on match length is *kr* (Haubold et al., 2009). In *kr*, the average match length is computed to estimate the distance between sequences. The idea of *kr* is based on the observation that matches are interrupted by mismatches, i.e. a mutation. Thus, for higher mutation rates, the expected match length decreases. For example, the expected match length for two random sequences and a substitution rate of 0.01 is 100, whereas the expected match length for a rate of 0.02 decreases to 50.

Anchor Distances

In Haubold et al. (2014) the concept of *anchor distances* is implemented. The method is based on the comparison between short and exact alignments. For anchor distances, maximal matches between two sequences are computed and termed *anchor* if they are maximal, unique and longer than a threshold. If the distance between at least two anchors is equally spaced on both sequences, the anchored regions are united to form an anchor pair. An example of the anchor pairs is shown in Figure 2.2, anchor pairs that are not equidistant are not considered (Haubold et al., 2014). The number of mismatches per site is estimated by counting the mismatches for the anchor pairs and the regions they cover. Using the Jukes-Cantor equation (Jukes and Cantor, 1969), the number of mismatches is used to estimate the evolutionary distance between both sequences (Klötzl, 2020).
The anchor distance metric is implemented in the tools *andi* and its faster successor *phylonium* (Klötzl and Haubold, 2019). In *andi*, the anchors are computed for each pair of sequences whereas *phylonium* calculates only the matches between a reference and each genome. Both, *phylonium* and *andi* performed well in a benchmark study for alignment-free methods (Zielezinski, 2019).

2.3 String Matching

Finding matches is an essential part of many alignment tools. The search for matches boils down to the question “Does a longer text T contain a shorter pattern P, and if so, where?” which we will refer to as the *exact matching problem*. We will look at some approaches to solve this problem in the past leading to the structure that was used for this thesis, namely the enhanced suffix array (ESA) (Abouelhoda et al., 2004).

The naïve way to find P in T is by comparing them for each position. We start at the first position in P ($P[0]$) and search for a matching character along T. If the first position of the pattern matches at a position i in T we continue to compare the subsequent characters in both strings, that is $P[1]$ and $T[i+1]$. The text T contains P if the complete query sequence can be processed that way and we report i as the starting position of P in T. If a mismatch is found the pattern does not start at $T[i+1]$. Either way we continue the comparison at the next position in T, $T[i+1]$ and again, the first character in P.

The naive approach is simple and easy to implement but has a worst case runtime of $O(|P| \times |T|)$ which makes this ineffective for large strings. The runtime can be improved if we can skip the comparison for characters where we might already know the result. To do so, many algorithms preprocess one of the inputs in some form. One famous approach to solve the exact matching problem is the algorithm developed by Knuth, Morris and Pratt (Knuth et al., 1977). The *Knuth-Morris-Pratt* algorithm effectively preprocesses the query to achieve an overall runtime of $O(|P|+|T|)$. The basic idea is that we do not have to always restart the comparison at the first character of P after a mismatch. The pattern can be shifted by more than one position if the location of the mismatch is known and the number of shifts can be determined by preprocessing the pattern (Gusfield, 1997). The *Knuth-Morris-Pratt* algorithm takes $O(|T|)$ in the worst case for the actual comparison and additional runtime of $O(|P|)$ for preprocessing each pattern. Other algorithms like the *Boyer-Moore* Algorithm (Boyer and Moore, 1977) can solve the matching problem in $O(|P|+|T|)$ as well.

However, a disadvantage of these approaches is that the time spend with preprocessing cannot be reused for other queries. If the goal is to match multiple queries on the same reference, the additional runtime for processing the reference is added each time for every query. Gusfield (1997) redefines this task as the *substring problem*. Here the text T is provided in advance. After some linear or $O(|T|)$ preprocessing time the exact matching problem must be solved in $O(|P|)$. In other words, for a known sequence one has to be able to search for patterns inside this sequence in time only dependent on the length of the pattern, not the text.

Suffix Tree

A structure that allows us to search efficiently inside a known sequence is the suffix tree (Weiner, 1973). The suffix tree has several applications, one of them is the substring problem (Gusfield, 1997) that can be answered in optimal, linear time. Abouelhoda et al. (2004) describe the suffix tree as "one of the most important data structures in string processing" especially for very large and not changing sequences like genomes. An example of a suffix tree is shown in Figure 2.3. For a given string T a suffix tree is a rooted directed tree that contains a leaf for every suffix of T. Every leaf is labelled with the starting index of said suffix. Starting from root, the path through the tree to a leaf corresponds to the suffix spelled out that belongs to that leaf (Haubold and Wiehe, 2006). The edges in this tree contain substrings of T. Common substrings of suffixes are aggregated into a common path. A node indicates a position where the substrings differ. Therefore, every node has

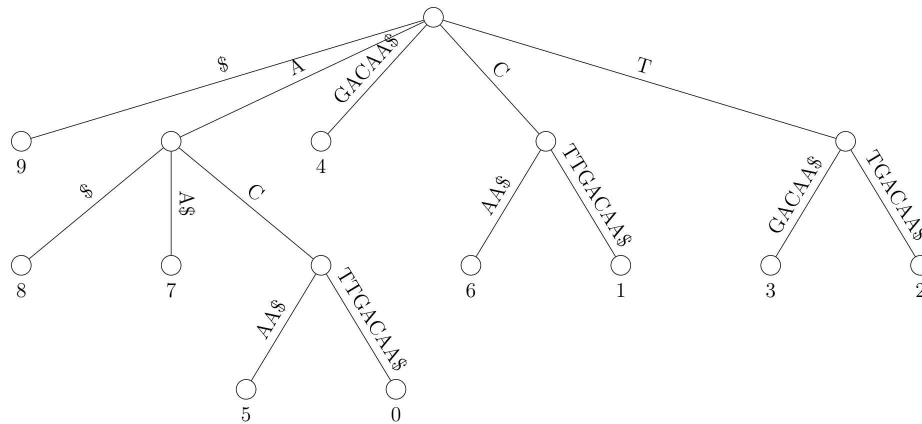


Figure 2.3: Suffix tree for S=ACTTGACAA$

at least two outgoing edges. This means that for each node in the tree the following leaves belong to suffices containing the text up to this node. Usually T is appended with a unique character (typically '$') that is not contained in the rest of T to mark its end.

To search for a pattern P we start at the root node and follow the outgoing edge that has the same label as the first character in P. This is repeated at each succeeding node and the next character in P. That way we can move along the tree. If we are able to process P completely, P is a substring of T. The labels of the leaf-nodes that are descendants of the current node contain the starting positions of P in T. If we reach a leaf or if no outgoing edge matches the next character in P we know that T contains the prefix of P up to the current position. For the second case, when we are still at an internal node, we know that the prefix of P is contained in T multiple times and retrieve the starting indices again by checking the descending leafs. This way it is possible to decide the *substring problem* in optimal time.

Fo all its undisputed ingenuity, the suffix tree is often criticized for its space consumption (Manber and Myers, 1993). Moreover, it is noted that despite its importance in theory it is not that common in practice (Abouelhoda et al., 2004). In comparison, the suffix array introduced by Manber and Myers (1993) has gained wider use.

2.4 (Enhanced) Suffix Arrays

The suffix array (SA) was introduced by Manber and Myers (1993) to provide a data structure similar to the suffix tree that is simpler and more space efficient. In its initial form, SA requires $4n$ bytes for $n = |T| < 2^{32}$ characters and can solve the substring problem in $O(|P| \log |T|)$ time using binary search. This can be further improved to $O(|P| + log\ |T|)$ using an additional table that contains the longest common prefix (LCP) (Manber and Myers, 1993). The suffix array of a string T is a sorted list or table that

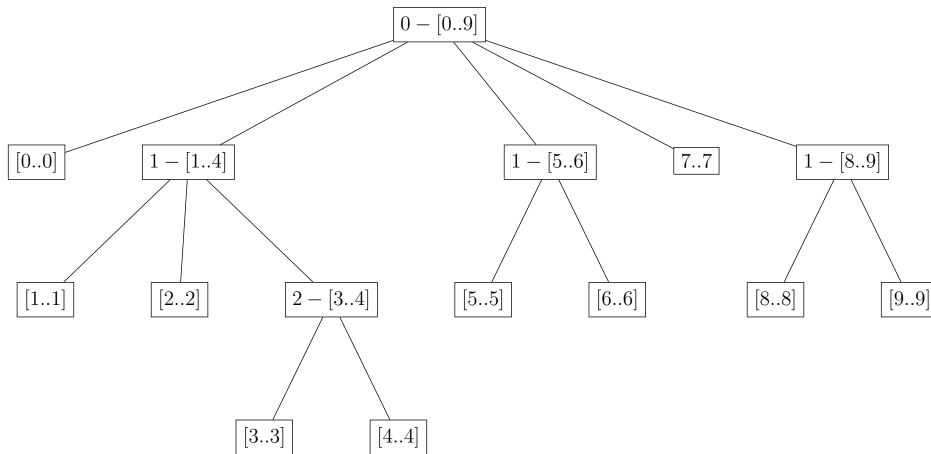


Figure 2.4: LCP-interval tree for S=ACTTGACAA: The indices at the nodes and leafs denote the positions at the ESA.

i	SA	LCP	CLD.L	CLD.R	S[SA[i]..]	lcp-intervals
0	9	-1	-	10	$	
1	8	0	-	5	A$	
2	7	1	-	3	AA$	1
3	5	1	-	4	ACAA$	2
4	0	2	-	-	ACTTGACAA$	0
5	6	0	2	7	CAA$	1
6	1	1	-	-	CTTGACAA$	
7	4	0	6	8	GACAA$	
8	3	0	-	9	TGACAA$	1
9	2	1	-	-	TTGACAA$	
10	-	-1	1	-	-	

Figure 2.5: Enhanced Suffix Array for T=ACTTGACAA

contains all suffices of T in alphabetical order. The LCP-array contains the length of the common prefix of an element with its predecessor in the sorted suffix array.

In Abouelhoda et al. (2004) the idea of the *enhanced suffix array* (ESA) is presented to make more use of the information contained in the LCP-array and *enhancing* the suffix array by adding additional tables. Henceforth, two concepts of additional tables, the *LCP-intervals* and the *child table* that allow to answer the substring problem in optimal linear time $O(n)$ will be shown. These concepts are based on Abouelhoda et al. (2004) and further developed in Ohlebusch (2013).

Using LCP-intervals it is possible to implicitly build a suffix tree or an equivalent structure at least, the *LCP-interval tree*. This LCP-interval tree is conceptual and does not have to be actually build but its structure resembles the initial suffix tree (see Figure 2.4). In a sense, a LCP-interval groups the elements of a SA that share a common prefix into

a subinterval. We define the LCP interval more formally as this is necessary for the definition of the child table.

Definition 2.4.1. (Taken from Ohlebusch (2013, Definition 4.3.1)) An interval $[i..j],\ 0 \leq i < j \leq n$ in an LCP-array is called LCP-interval of value l if and only if

1. LCP$[i] < l$,
2. LCP$[k] \geq l$ for all k with $i + 1 \leq k \leq j$,
3. LCP$[k] = l$ for at least one k with $i + 1 \leq k \leq j$,
4. LCP$[j + 1] < l$

We also define an index l-index if LCP$[k] = l$ for $i + 1 \leq k \leq j$ for interval $[i, j]$. The l-indices define the shortest LCP and therefore a local minimum of their l-interval. They set the level of this interval and the nodes of the LCP-interval tree. For example in the interval [1..4] from Figure 2.5 all suffices start with A and are on level 1, they also share a common node in Figure 2.4.

The local minima inside the LCP-array indicate boundaries between LCP-intervals and hence different paths or sections on the LCP-interval tree. To be able to traverse the LCP-interval tree top-to-bottom similarly to the search on a suffix tree we need an additional table, the *child array* (Ohlebusch, 2013). The child array contains the information of two pointers CLD.L and CLD.R, a left and a right one respectively. We define CLD.L and CLD.R according to Ohlebusch (2013, Theorem 4.3.25):

1. If LCP[i] $\leq$ LCP[j+1], then CLD[j+1].L stores the first l-index of the LCP-interval $[i..j]$
2. If LCP[i] $>$ LCP[j+1], then CLD[i].R stores the first l-index of the LCP-interval $[i..j]$

In other words CLD.L and CLD.R point to the first local minimum of the interval on their left or right side respectively. CLD represents a tree that recursively splits the suffix array at the local LCP minima in two (Ohlebusch, 2013). This can also be illustrated by a simple conceptual algorithm to construct the child array presented in Frith and Shrestha (2018). This algorithm is shown in Listing 2.1. It rather helps to explain the idea behind the child array and is not indented to be efficient.

The pointers in the child array can be considered to be similar to the edges in the suffix tree that allow a guided search through the (virtual) LCP-interval tree. A complete traversal of the suffix array from Figure 2.5 would start at position 0 which points, by definition, to the last element of the suffix array at position 10 (step 1). Since this is the last element of the array it only has a left child at $i = 1$ that points to its right sibling with LCP=0, that is position 5 (2). At position 5 there are two pointers. On the left hand to the a-interval with $LCP = 1$ and to the right hand where we are still on the interval with $LCP = 0$. CLD[5].L points to the first minimum of [2..5] which is 2 (3). CLD[2].R points to position 3 and 4 (step 4 and 5). CLD[5].R points (3 as well) to its right sibling at position 7 that

```
1 fn makeChildTable
2 requires startPos, endPos, idx, LCP, CLD
3 if endPos − startPos < 2
4     return
5 end
6 mid ← argmin(LCP[startPos+1..endPos])
7 CLD[idx] ← mid
8 makeChildTable(startPos, mid, mid−1)
9 makeChildTable(mid+1, endPos, mid)
```

Listing 2.1: Simple recursive algorithm to construct the child table, adapted from Frith and Shrestha (2018)

points to position 8 (4-5). That way the complete tree can be traversed in five steps at most.

To save memory, CLD.L and CLD.R can be merged into one single array by placing CLD[i].L at CLD[i-1].R without overriding information. If there is a pointer at CLD[i].L, CLD[i-1].R is always empty because CLD[i].L points to the minimum of the interval that ends at i-1. Since the boundary between the two intervals must be between position i-1 and position i there cannot be any pointer at CLD[i-1].R we can put CLD[i].L there.

2.5 Suffix Array Construction

Using the ESA, the searching time can be reduced to $O(|P|)$, making the search independent of the size of the text. For this to work for the *substring problem*, the initial suffix array needs to be constructed. It is possible to convert between suffix arrays and suffix trees and vice versa in linear time (Ohlebusch, 2013). For the sake of completeness it should be noted that constructing the suffix tree first and using this to build the SA leads to a hypothetical linear solution since the suffix tree can be constructed in linear time as well (Gusfield, 1997). But this misses the point of the suffix array, which is to replace the extensive space consumption of suffix trees. Also it is possible to naively build a suffix array using any sorting algorithm. To order the suffixes their first characters are compared. If they differ, sorting is possible, otherwise their next characters are compared. For a text of size n, the runtime of the naive approach can be estimated by expecting that each comparison takes $O(n)$ in worst case. The overall runtime depends on the sorting algorithm but the commonly used *quicksort* with an expected complexity of $O(n \log n)$ would result in an overall runtime of $O(n^2 \log n)$ for example.

Specialized suffix array construction algorithms achieve significant improvements by taking into consideration that the elements to be sorted are related and nested. When Manber and Myers (1993) published their concept of the SA, they applied a technique called *prefix-doubling* based on Karp et al. (1972) to reduce the runtime to $O(n \log n)$ for a text

of size n. Here the suffixes are sorted into buckets according to their first character. Successively these buckets are sorted for twice the number of leading characters, hence the name prefix-doubling. Sorting according to the first character can be done in $O(n)$ in at most $O(log_2\, n)$ iterations so this approach takes about $O(n \log n)$ (Puglisi et al., 2007). In the early 2000 several approaches to construct the SA in $O(n)$ were introduced (Ko and Aluru, 2003; Kim et al., 2003; Kärkkäinen and Sanders, 2003), which are reviewed and surveyed thoroughly in Puglisi et al. (2007).

Most effective algorithms then and today are based on the idea of induced sorting (Xie et al., 2020). The idea of these algorithms is to sort a well selected subset of suffices first. The resulting order can be used in subsequent steps to determine the order of the remaining suffices (Ohlebusch, 2013; Dhaliwal et al., 2012). These algorithms use *divide-and-conquer* (and merge) strategies for sorting the suffices. The key difference between the approaches is the selection criterion for the subsets, which as a consequence effects the conquering, inducing and merging process as well (Shrestha et al., 2014). For example, the *Skew*-Algorithm by Kärkkäinen and Sanders (2003) selects the suffices S of a text T as follows:

$$S = \{T_i ... | i \mod 3 \neq 0\}$$

The suffices in S are associated with 3-grams containing the first three characters of their suffix. The 3-grams can be sorted in linear time using a *radix*-sort (Ohlebusch, 2013). In the next phase, the remaining elements with $S = \{T_i ... | i \mod 3 = 0\}$ are sorted using the information gained from sorting the first step. If two comparing characters equal, it is sufficient to compare the ranks of their direct successors.

Another algorithm, the SA-IS (suffix array induced sorting) algorithm introduced by Nong et al. (2011) is the designated best-known linear-time algorithm in both theory and practice (Timoshevskaya and Feng, 2014; Shrestha et al., 2014). Here, the suffices are classified into S-type and L-type suffices, depending on whether they are smaller (S) or larger (L) than their right neighbouring character. In addition the LMS-suffices (leftmost S-type) are identified. The LMS suffices are sorted and used to induce the order of the L-type suffices. The SA-IS algorithm is based on the algorithm by Ko and Aluru (2003) who proved that the order of L-type suffices can induce the order of the S-type suffices (Ohlebusch, 2013).

It should be noted that in practice algorithms with worse than linear complexity can perform better than linear algorithms (Puglisi et al., 2007). The program often referred to as 'best in practice' algorithm (Timoshevskaya and Feng, 2014) is *libdivsufsort*[1] with a worst-case run time $O(n \log n)$, reviewed by Fischer and Kurpicz (2017). More recently a new implementation of the SA-IS algorithm, *libsais*[2], seems to outperform *libdivsufsort*. The author states that this algorithm is still based on SA-IS but improves its performance by better exploiting recent hardware developments [3].

[1]https://github.com/y-256/libdivsufsort, accessed 08.09.2022
[2]https://github.com/IlyaGrebnov/libsais, accessed 08.09.2022
[3]https://encode.su/threads/3579-New-saca-and-bwt-library-(libsais), accessed 11.09.2022

3 Implementation

In this section we will take a look at the implementation of the alignment-program *par* (Pile Anchors on Reference) and explain the algorithms used. The objective of *par* is to return the explicit anchor alignments that stay implicit in *phylonium*. *Phylonium* computes the suffix array for a single reference sequence and piles all other queries on the reference. The alignments with the query that overlap among the reference are used for pairwise distance computation. Similar to *phylonium*, I start in *par* with building the enhanced suffix array for the reference sequence. In a next step, the queries and the reference are compared. In contrast to *phylonium*, the resulting alignments are not used for distance computation. Instead, the aligning segments are piled on the reference to build blocks of multiple sequence alignments. These blocks are further split and printed to the multiple alignment format MAF.

Par is implemented in Go Programming Language. Most parts were written in literate programming style (Knuth, 1992) using noweb (Ramsey, 1994). *Par* is available at

https://github.com/dadidange/par_lp.

The program roughly consists of the following steps:

1. read the input sequences,
2. build the ESA for the reference,
3. find the anchors in queries and reference,
4. filter for orthologies (optional, to fit *phylonium*)
5. pile the anchors
6. print the alignment

3.1 Input and Preprocessing

The idea of *par* is to work on Unix-based systems like Linux on the command line using text-files as input. Apart from some options that are not mandatory, the program expects DNA-sequences in Fasta-format. Fasta-files are plain text files and widely used in biology and bioinformatics. They can contain single or multiple nucleotide sequences. Each sequence starts with a header or definition line that begins with a greater that ('>') character, which is followed by an identifier and optional comments. Subsequent lines up

to the next ‘>’ contain the actual nucleotide characters. There is no mandatory length for the sequence lines, but it became convention to insert a line break after 70 or 80 characters, depending on the definition. For *par*, the header is reduced to only contain the sequence identifier. The actual sequence is checked for characters that are not ‘ACGT’ that are replaced by an ‘N’.

Par needs a reference to build the ESA for. The reference is picked either by the user or a sequence of median length is picked. This choice has proven to be more robust to outliers than choosing extremes in Klötzl (2020). I exploited other criteria for picking references, for example sequences that minimize their total evolutionary distance to other sequences, but this had little or no effect on the results.

3.2 ESA revisited

The enhanced suffix array (ESA) is the central data structure of the program. While its concept was explained in the previous section (see Section 2.4), the algorithms for its application and implementation are discussed below.

By default, the ESA is calculated for both strands of the reference sequence. For this purpose, the reverse complement is appended at the end of the initial sequence, separated by a ’#’ so that the matches cannot span both strands.

Constructing the ESA The ESA as we use it contains three tables or arrays, namely the sorted suffix array, the LCP-array and the child array. For *par* the suffix array can be build using three different libraries, *libdivsufsort* and the suffixarray package[1] in the GO standard library (GoSais). Libdivsufsort is one of the most widely used libraries and is used in *phylonium* (Klötzl, 2020). Libsais was published more recently, later than *phylonium* and is said to outperform *libdivsufsort*. We benchmarked these approaches in Section 5.1.

Based on the suffix array, the LCP-array is build using the Kasai algorithm (Kasai et al., 2001).

Child Array The algorithm to construct the child array from the LCP-array is based on Ohlebusch (2013, Algorithm 4.11) and is shown in Listing 3.1. It loops through the LCP-array and adds its elements to a stack. A decrease in the LCP-array indicates a local minimum, that is, a node in our tree. Therefore, a right child, left child or both are added at these positions. Building the child array takes again $O(|T|)$ (Ohlebusch, 2013) and completes the construction of the ESA for our purposes.

[1] https://pkg.go.dev/index/suffixarray

```
1   fn buildCld
2   requires LCP
3   let n ← |LCP|
4   push(0) // store indices in a stack
5
6   for k ← 1 to n+1 do
7       while LCP[k] < LCP[top()] do
8           last ← pop()
9           while LCP[top()] = LCP[last] do
10              CLD.R[top()] ← last
11              last ← pop()
12          end
13          if LCP[k] < LCP[top()] then
14              CLD[top()].R ← last
15          else
16              CLD[k].L ← last
17          end
18      end
19      push(k)
20  end
21  output CLD
```

Listing 3.1: Construction of the child array, Ohlebusch (2013)

3.3 Finding the Anchors

The ESA allows us to effectively find anchor pairs or groups between the queries and the chosen reference sequence. This process is based on three algorithms, *GetInterval*, *GetMatch* and *AnchorMatches*. The function that operates directly on the ESA and the child array in particular is *GetInterval*.

GetInterval returns the subinterval or child-interval of a given interval i that starts with a character c. Its pseudo-code algorithm can be seen in Listing 3.2 which is modified from Ohlebusch (2013) and Klötzl (2020). *GetInterval* starts by looking for singletons, i.e. intervals that only contain one element. If that is not the case it iterates across the child array looking for the interval that starts with c. The algorithm shown is different to the textbook approach in one small detail, the loop starting at line 17. This adaption considers that two intervals might end at the same position in the suffix array. Given a child interval $\{i, j\}$ that is the last child interval of its parent $\{h, j\}$ with $h < i < j$, we cannot take CLD[j] to find the first minimum in $\{i, j\}$. CLD[j] might point to the an other local minimum of the larger interval that lies outside $\{i, j\}$. Instead we use CLD[h] that points to the next minimum of $\{h, j\}$, i.e. a right sibling or child. Since $\{i, j\}$ must be a

child of $\{h, j\}$ we can follow the right pointers until we will eventually arrive at the first local minimum inside $\{i, j\}$.

GetMatch is the function that calls *GetInterval* and enables us to find matches between our reference and any query sequence. More precisely, it returns the longest prefix of a query that matches any suffix of the reference. *GetMatch* calls *GetInterval* once per character at most and tries to extend the matching characters for the length of the LCP in that interval to save unnecessary calls. If the LCP and the following characters still match, *GetInterval* gets called again in the next iteration with the new interval and the remaining characters in the query. *GetMatch* runs in $O(\sigma|P|)$ time with σ as the size of the Alphabet. Since our alphabet is constant ($\sigma = 4$) it runs in $O(|P|)$ time (Ohlebusch, 2013).

AnchorMatches The anchoring step is the last one in the process of finding matches. The corresponding algorithm is adapted from (Klötzl, 2020, Listing 3.1) and can be seen in Listing 3.4. The algorithm finds a list of regions that are homologous in the reference and a query sequence by calling *GetMatch* for every suffix from the query. As a result, it always finds the maximal match starting at each position. If the criteria for minimum length and uniqueness apply, the distance to the previous anchor gets examined. If the matches on both, query and reference sequence are equidistant and on the same strand the region and both anchors are merged into a single new region of anchor pairs. These anchor regions approximate local, gap-free alignments that start and end with exact matches. Recall the anchors that form a pair from Figure 2.2. Both anchors are assembled into one local alignment that we assume to be homologous. In further functions, the region as a whole is processed rather than the individual anchors.

Anchor Threshold In *phylonium* and *par* the default threshold for minimum length is defined by the following equation which is taken from Haubold et al. (2009, Eq.4).

$$P(X^* > x) = 1 - \sum_{k=0}^{x} 2^x \binom{x}{k} p^k \left(\frac{1}{2} - p\right)^{x-k} \left(1 - p^k \left(\frac{1}{2} - p\right)^{x-k}\right)^{|S|}$$

This describes the probability that a common substring or shustring (Shortest Unique Substring) (Haubold et al., 2009) is larger than x for two unrelated sequences with length $|S|$. The threshold can also be defined by the user.

3.4 Sorting and optional filtering

The function *AnchorMatches* returns a list of local pairwise alignments between query and reference sequence of which we say that they are homologous. These alignments are

arranged in the order of their matching position at the query sequence. We sort them in their appropriate order in the reference sequence, as this is needed in subsequent steps.

The first step in which this is required is optional and comes from *phylonium*. Due to duplication events it is possible that multiple segments within a sequence overlap on the reference. In *phylonium*, such overlapping segments are removed and only one is left since multiple comparisons of the same sequence would bias the distance estimation. Only the element that belongs to the chain that aligns the maximum number of nucleotides is kept. The best chain is calculated by iterating through the sorted alignments. For each sequence the predecessor with the highest score that does not overlap is identified and the own score is set to $predecessor.Score + own.Score$ with score being the number of aligning nucleotides. Subsequently the sequence with the highest score and its predecessors are chosen to be the elements that are further considered for comparison.

This step is optional for *par* as the ability to align duplications might be desirable for whole genome alignments. However, the option to remove them is kept to produce an alignment that better corresponds to phylonium. We compared both approaches in Section 5.3.

3.5 Pile Anchors on Reference

The next step is to find overlapping segments between sequences to group them into alignment blocks. We consider one block to consist of different anchors that overlap with their matching positions on the reference. A block is framed by the earliest starting and latest ending index on the reference sequence of its contained alignments. The algorithm for piling the alignments into alignment blocks is shown in Listing 3.5. We have a list that contains a list of alignments for each sequence. Furthermore, we have a list of pointers indicating which alignment to check next for each sequence. First, we pick the sequence with the earliest start and add it to the block. Then we iterate through the pointers and compare the indices of the alignment it points to with the block coordinates. The block coordinates are the start of the earliest alignment and the end of the alignment furthest to the back. We check the alignment positions on the reference and pile the alignments on the reference when they overlap. Figure 3.1 depicts the piling process for two sequences. If an element is added, the pointer is incremented to point to the next alignment on that sequence. The alignments are sorted by their starting positions on the reference. This allows them to be processed in their order in the list without the possibility to overlook an alignment. If the alignment pointed to by the pointer does not overlap with the current region, the subsequent alignment does not either, because it starts later than its predecessor. The piling process is repeated until no further element can be added to the block. If there are still alignments left, a new block is added and the remaining alignments are processed in the same way.

When all alignments are processed, each block is divided into multiple printing blocks to comply with the output format MAF.

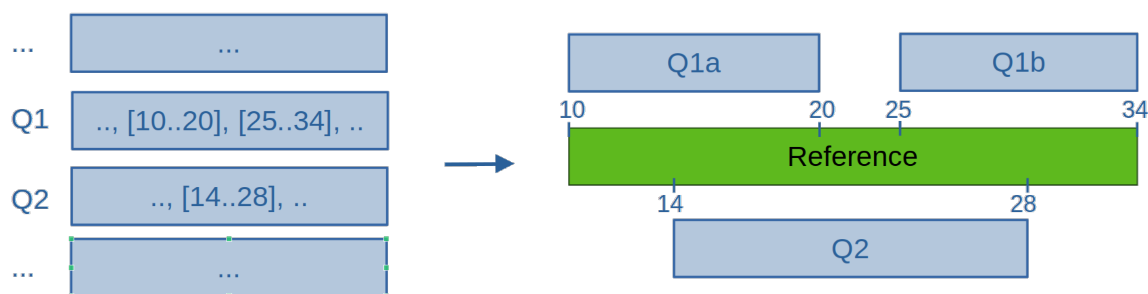
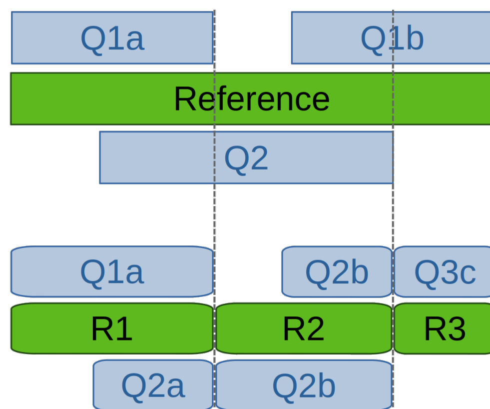


Figure 3.1: Example of the piling process. On the left, the list of alignments that stores the alignment coordinates with respect to the reference sequence. On the right, how the overlapping alignments are piled on the reference.

MAF We chose the Multiple Alignment Format (MAF[2]) as output format. MAF is a simple line-oriented text based format used by many alignment tools (Armstrong et al., 2019). MAF was used as well in the Alignathon project (Earl et al., 2014) for benchmarking MSA tools. MAF-files can store a series of multiple sequence alignments in a separate block for each alignment. Each sequence inside a block is on a single line that contains additional information in this order: name of the sequence, zero based starting position of the current alignment in the sequence, size of the alignment without gaps, strand direction, size of complete source sequence and finally all aligning nucleotides including possible gaps. Optional lines for additional information can be added as well. Apart from it being widely used, another advantage of MAF is that it contains more information than other alignment formats like Fasta or Phylip. This makes it easier to convert downwards to one of these formats while conversion in the other direction is not possible. For an example of our interpretation of the MAF format see Figure 3.2b.

Since the official format definition lacks some details about how gaps and orthologies are handled, we followed the specification of Dutheil et al. (2014, Figure 1). No alignments are added consecutively to a block, as this could lead to gaps in the reference sequence. Whether these gaps can also lead to further gaps in alignments of other sequences is not always predictable, if the sequences are not to be run through several times. Therefore, an alignment block is split into multiple synteny blocks according to the ends of the alignments it contains. The first block starts at the start of the first aligning position and ends at the earliest end of the alignments it contains. The remainder of the alignments that go beyond that position is added to a new block as well as some possible new alignments that overlap within that next segment. As a result, an alignment can only contain gaps at the beginning and not at trailing positions. Figure 3.2 depicts how the alignment from Figure 3.1 is split into multiple blocks.

[2]https://genome.ucsc.edu/FAQ/FAQformat.html#format5



(a) Piled blocks (top) that are split for printing (bottom).

```
##maf version=1 scoring=none
#optional comment
a
s R  10 10 + 47 ACGGCTCTAA
s Q1 4  10 + 43 ACGGCTCTAA
s Q2 12  6 + 38 ----CTCTAA

a
s R  20 8 + 47 GTTATACA
s Q2 18 8 + 38 GTTATACA
s Q1 17 3 + 43 -----ACA

a
s R  28 6 + 47 GGTCCA
s Q2 20 6 + 34 GGTCCA
```

(b) MAF output generated by *par* (exemplary illustration). The values in blue are arbitrarily chosen and cannot be derived from the right figure.

Figure 3.2: Split alignment blocks and print to MAF: (a) depicts how a single alignment block (top) is split into multiple blocks. The MAF file in (b) shows how the blocks could be displayed.

```
1  fn GetInterval
2  requires S, SA, CLD, LCP // ESA
3  input [i..j] //Interval to check
4  input c //character to check
5
6  if i = j:
7      if S[[SA[i]]] = c:
8          output [i..j]
9      else
10         output ⊥ //empty interval
11     end
12 end
13
14 //find interval boundaries
15 lower ← i
16 upper ← CLD[j]
17 while upper ≤ lower:
18     //iterate through children
19     upper ← CLD[upper]
20 end
21
22 l ← LCP[upper]
23
24 while LCP[upper] = l:
25     if S[SA[lower] + l] = c
26         //found match
27         output [lower..upper−1]
28     end
29     //increment interval boundaries
30     lower ← upper
31     if lower = j:
32         break
33     end
34     upper = CLD[upper]
35 end
36 //final check for last child interval
37 if S[SA[lower] + l] = c:
38     output [lower..j]
39 else
40     // no suiting child intervals for c
41     output ⊥
42 end
```

Listing 3.2: Method GetInterval, based on Ohlebusch (2013) and Klötzl (2020)

```
 1  fn GetMatch
 2  requires S, SA, CLD, LCP // ESA
 3  input Q
 4
 5  in ← [0..|S|]
 6  k ← 0
 7
 8  while k < |Q|:
 9      [i..j] ← GetInterval(in, Q[k])
10      if [i..j] = ⊥:
11          if k = 0:
12              output ⊥
13          else
14              output in
15          end
16      end
17      l ← min(|Q|, LCP[j])
18      //compare prefix
19      for p = k to l:
20          if S[SA[i] + p] != Q[p]:
21              output [i..j]
22          end
23      end
24      k ← l
25      in ← [i..j]
26  end
27  output in
```

Listing 3.3: Method GetMatch, based on Ohlebusch (2013) and Klötzl (2020)

```
1   fn AnchorMatches
2   requires esa, threshold
3   input query
4
5   last ← ⊥ // empty match
6   homologies ← ⊥ // empty list of homologies
7   current ← homology(0,0) // starting homology
8
9   while q < |Q|:
10      match ← getMatch(esa, Q[q..])
11      if isAnchor(match):
12          if q − lastQ == match.start − last.start and match.strandDir == last.strandDir:
13              //form pair if equidistant and on same strand
14              current.len = q + m.len − current.start
15          else:
16
17              homologies.append(current)
18              current ← m
19          end
20          lastQ ← q
21          last ← match
22      end
23      q ← q + max(1, match.len) //advance in query
24  end
25  output homologies
```

Listing 3.4: Method AnchorMatches, based on Klötzl (2020)

```
1   fn pileBlocks
2   input H //list of sorted lists of homologies
3   requires b //empty block
4
5   n ← size(H)
6   next ← zeros(n)//pointer to next element for each sequence
7
8   s ← argmin_i(H_i[0].refStartIdx)
9   b.Add(H_s[next_s])
10  next_s ++ // point to next in sequence
11  while added:
12     added ← false
13     for i = 0 to n:
14        if H_i[next_i].overlaps(b):
15           b.add(H_i[next_i])
16           next_i ++
17           added ← true
```

Listing 3.5: Method pileBlocks

4 Evaluation Method

In this chapter, I introduce my approach to evaluating the results of *par*.

Apart from *par*, we also tested the alignment tools *mugsy* (Angiuoli and Salzberg, 2010) and *sibeliaZ* (Minkin and Medvedev, 2020). *Mugsy* is a popular tool for fast multiple alignment that performs well for closely related genomes. On more distantly related genomes, however, it is outperformed by *cactus* (Armstrong et al., 2020) in Earl et al. (2014). *SibeliaZ* is a more recent alignment tool that was also developed for closely related genomes and showed good results alongside *cactus* (Minkin and Medvedev, 2020). Both tools, *sibeliaZ* and *cactus* seem to require memory that exceeds the range of personal computers. We decided for *sibeliaZ* and against *cactus* for additional evaluation, since *cactus* is even more memory extensive and did not terminate in some experiments conducted in the *sibeliaZ* paper (Minkin and Medvedev, 2020).

Evaluating whole genome alignments is difficult since the true alignment and evolutionary history is rarely known (Dewey, 2019). Compared to global alignments, scoring metrics like the *sum-of-pairs* score cannot be considered. The *sum-of-pairs* method returns the sum of pairwise scores for all pairwise alignments (Batzoglou, 2005). This scoring scheme cannot be used here because whole genome alignments can consist of multiple blocks. They take into account that genomes might consist of highly conserved regions that share homology that are interrupted by regions that are not related at all. Therefore, the rating could be artificially improved by skipping positions that contain mutations and result in low scores.

To work with a ground truth, many studies use simulated data to evaluate their results. The *Alignathon* project (Earl et al., 2014), which is one of the largest and most cited benchmark study for whole genome alignment, relied on simulated data for most of their experiments as well (Dewey, 2019). Unfortunately, they also relied on large server infrastructures, so their results are not reproducible on standard personal computers. Also, they only evaluated accuracy and did not measure time consumption. For the *Alignathon* some tools ran on large servers, e.g. *cactus* on a 64 core machine with 1 terabyte of RAM for over 500 hours. Again, this makes these results difficult to reproduce on personal computers. What we have taken from the Alignathon study, however, is the scoring method using the *mafComparator* tool for scoring MAF files.

4.1 Scoring Approach

The *mafComparator* estimates *precision* and *recall* to measure performance. This tool compares two MAF files by performing homology tests as follows. Given two sets of pairwise alignments A and B a pair of positions in A is picked. The homology test returns true if the pair picked exists in B and false otherwise. *MafComparator* runs twice through each MAF file to count the number of overall pairs in the first run and to perform the sampling process in the second run. If we consider one MAF file to be the 'truth' T and the other to be a 'prediction' P we can evaluate the alignment accuracy of P by calculating the measures of precision and recall.

$$precision = \frac{T \cap P}{P},\ recall = \frac{T \cap P}{T}$$

Precision indicates the ration of correct alignments among the predicted alignments by P and recall defines the ratio of true alignments predicted in P among all alignments. Both measures are often summarized using their harmonic mean to form the F-measure or F_1-Score that is defined as

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In addition to the form of accuracy described above, we chose to measure time and memory consumption to measure performance. These three categories can provide a good reference for possible use cases since the optimal alignment result in terms of accuracy is often infeasible. While we are confident about the time and space efficiency of *par* we estimate it to have a worse accuracy compared to other tools, since it originates from an alignment-free approach. This way, we can put the loss of accuracy in perspective to a gain in speed.

We measured time and space consumption using the 'time'-command for Unix operating systems. For memory consumption we measured the *maximum resident set size*. For time we summed the values for CPU-seconds the process spend in user-mode and in system(kernel)-mode. We ran all tests under a Intel Xeon W-2245 CPU clocked at 3.90GHz with 32GB RAM available.

4.2 Data

Many tools (Darling et al., 2010; Paten et al., 2008; Angiuoli and Salzberg, 2010; Armstrong et al., 2020; Minkin and Medvedev, 2020) use simulated data for evaluation since they can provide some kind of 'true' alignment. A disadvantage of this method is that the models on which the simulations are based on are only assumed to be correct and are often an abstraction of reality Chatzou (2016). Another way to evaluate alignments can be to compare them to other alignments calculated with other tools. An advantage of

this approach is that the results can be compared directly and put into perspective. This is particularly useful, as we again expect *par* to perform worse in terms of accuracy but faster and more efficiently. In direct comparison with other results, we can assess how much can be achieved with a given saving of resources.

We tested the performance on simulated data to examine the behaviour of *par* for different settings and on empirical data to compare its results to other tools.

4.2.1 Simulated data

Different properties of the input data can influence the results of the performance. Using synthetic data the influence of these properties can be examined individually and as isolated as possible. We wanted to check the change in performance with regard to changing mutation rates, an increasing number of genomes and an increasing genome size. The change in mutation rate is expected to influence the outcome of the alignment results in terms of accuracy whereas the increasing genome size and number of genomes should influence the time and memory consumption. The synthetic data was created with the ms tool (Hudson, 2002) that generates DNA sequences according to the Wright-Fisher-Model.

4.2.2 Real data

Although tests on simulated data are helpful to estimate fields of applications for the alignment tools and their limits it is important to find out to what extent this applies to reality. Due to the absence of known truths we decided to evaluate accuracy against the alignments output by *mugsy*. From this, the task results in how much of *mugsy*'s results can be obtained with *par*.

We used two datasets for real data.

Escherichia Coli We chose to evaluate the tools on a set of 29 whole *Escherichia coli* (E. coli) genomes that were previously used for *phylonium* and other benchmarks (Haubold et al., 2014; Leimeister et al., 2018; Angiuoli and Salzberg, 2010). The genomes have an average length of 4.9Mbp and an average substitution rate of 0.002.

Covid We also downloaded a set of 619 covid genomes from Germany on NCBI (as of August 2022) and randomly sampled 100, 200 and 300 respectively. This aims to test the performance of *par* on outbreak data consisting of many, closely related genomes. Compared to the E. coli genomes the covid genomes are more closely related with a substitution rate of 0.0002 and over 150 times shorter with about 30Kbp per sequence.

5 Results

5.1 Benchmarking Suffix Array Libraries

We benchmarked four different approaches to suffix sorting, the C-libraries *libdivsufsort* and *libsais*, the *suffixarray* package in the GO standard library (GoSais) and a naive sorting algorithm. We created three different random DNA sequences with 50kbp, 5Mbp and 50Mbp length respectively and took an arbitrary 5Mbp sequence out of the Eco29 dataset. As shown in Figure 5.1 *libsais* outperforms the other libraries with the naive approach being at least 10 times slower. *Libdivsufsort* is slightly quicker than GoSuf on the largest random sequence and the E. coli sequence. So GoSuf seems to be competitive in terms of suffix array construction. Unfortunately, the data type returned by GoSais is not compatible with the calculation of the ESA, for instance the actual suffix array is not made public. This makes the use of GoSais impractical, in particular since the two C libraries are still faster.

Figure 5.1: Average runtime in Milliseconds over 10 runs

dataset	Sais	DivSufSort	GoSais	Naive
rand-50K	1.47	5.78	5.60	17.37
rand-5M	211	298	292	2012
rand-50M	2097	3779	4299	31204
e. coli	240	318	379	2873

5.2 Simulated data

Increasing Mutation Rates

As a first test, we evaluated the alignment quality as a function of mutation rates. Increasing mutation rates should increase the difficulty of finding a suitable alignment. We computed alignments for 10 sequences with 30kbp each. This was repeated for an increasing ratio of segregating sites, starting from 150 (0.5%) up to 6000 (20%). We took the average results over 10 runs.

Figure 5.2 shows the accuracy of *par*, *mugsy* and *sibeliaZ* over an increasing number of mutations. For small mutation rates the tools achieve comparable accuracy while *mugsy* performs the best. For a mutation probability higher than 10% both, *par* and *sibeliaZ*

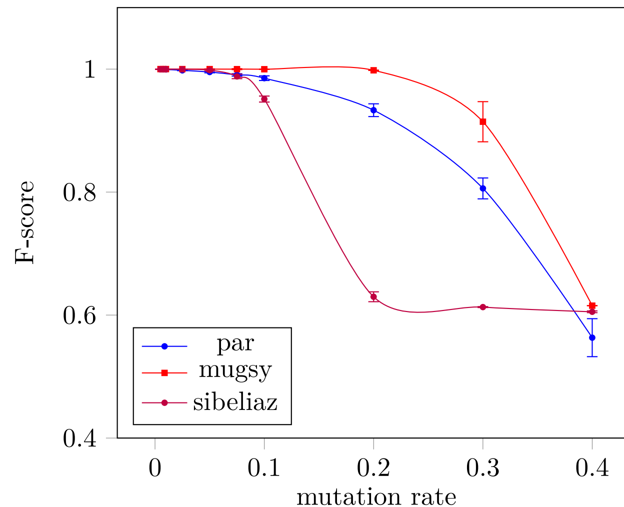


Figure 5.2: F-scores as a function of mutation rate on simulated data.

start to drop quickly. The curvature in *par*'s accuracy follows a similar curve as *mugsy*'s accuracy, but it starts to drop earlier. *SibeliaZ* drops heavily between a mutation rate of 0.1 and 0.2 but remains constant at about 0.6% accuracy.

To benchmark resource consumption we added *phylonium* as well, even though it does not produce actual alignments. This serves as an orientation for *par*, anything other than a similar scaling would be surprising.

Time and space consumption as a function of mutation rate are shown in Figure 5.3. For these comparably small and few sequences *par* is the fastest of the tools compared. It is around 10 times faster than *phylonium*, 100 times faster than *mugsy* and at least 250 times faster than *sibeliaZ*. For mutation rates up to 0.1% *mugsy* and *par* have similar memory consumption. However, simultaneously to the drop in accuracy the memory consumption of *mugsy* starts to rise and drop again. *Par* and *phylonium* never use more than 10 MB of RAM, *mugsy* never more than 50 MB. Space consumption of *sibeliaZ* is worse by orders of magnitude since depending on the mutation rate, 2000-3500 MB of RAM is used.

Par and *phylonium* have similar time spend on the CPU and are more than 20 times faster than *mugsy* or *sibeliaZ*. Also, *par* and *phylonium* are constant over increasing mutation rates in both, time and memory consumption. *SibeliaZ* consumes up to a third less time for increasing mutation rates whereas *mugsy* needs more time and memory for increasing mutation rates.

If we concentrate on *par* and *phylonium*, *par* is slightly faster at the expense of higher memory consumption than *phylonium*. This might be due to further processing of the alignments after the calculation of the ESA. To count mismatches, *phylonium* needs to compare sequences at the nucleotide level. Although the code is highly optimized, all overlapping alignments must be pairwise compared. This takes more time than the piling process in *par*, where only the indices need to be compared. However, in *phylonium* only

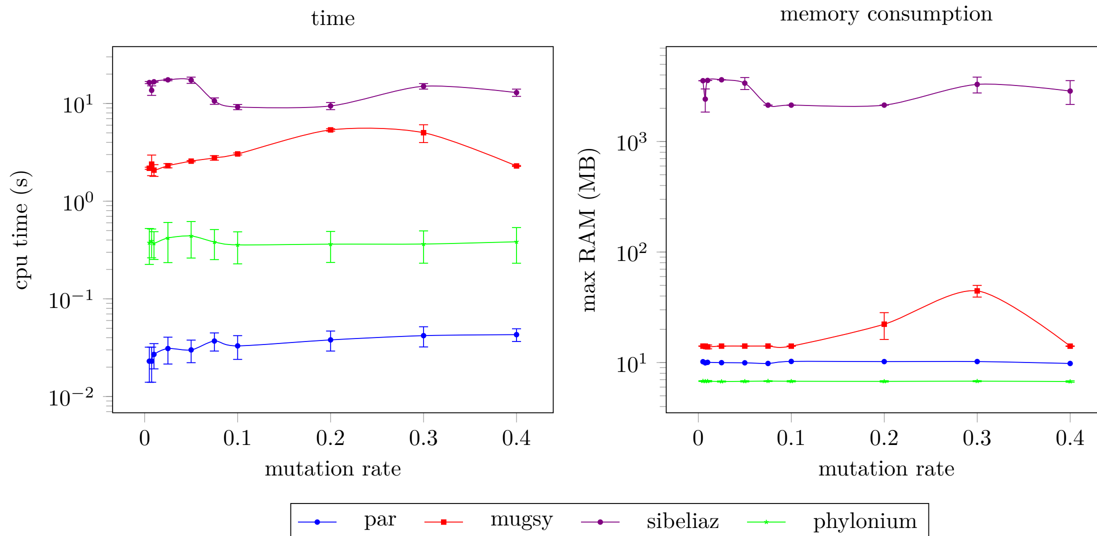


Figure 5.3: Time (left) and memory (right) consumption as a function of mutation rate.

two alignments have to be held in memory at the same time, whereas in *par* all aligning elements are kept in memory simultaneously.

Increasing Size and Number of Genomes

Next we evaluate the resource consumption as a function of genome size or and sample size. We evaluated genomes with sequences created using *ms* with about 2% segregating sites. We created genomes with length ranging from 10kbp to 5Mbp. For each length we had 10 genomes to be aligned and measured the mean performance over 10 runs. For the dataset with an increasing number of genomes to be aligned we created 10 to 200 sequences per test with a genome length of 30kbp each. All tools produced alignments with over 99% accuracy when they were able to finish, except for *sibeliaZ* on 200 genomes. *Par* and *phylonium* finished all tests within seconds.

For increasing genome length, *phylonium* and *par* produce similar runtimes, being almost one hundred times faster than *mugsy* and *sibeliaZ*. *Phylonium* consumes the least memory and *par* and *mugsy* scale very similar with increasing genome length. *Par's* alignments score not very differently than *mugsy's*, both ranging above 99% accuracy. Unfortunately, *sibeliaZ* does not seem to react well to the simulated data and is unable to produce reasonable alignments for this type of genomes longer than 100kbp on our machine.

Mugsy does not terminate on any run on our machine for more than 100 genomes even though the peak memory consumption is similar to *par*'s. SibeliaZ produces an alignment with only 58% accuracy. At the same time, its time and resource consumption drops slightly. It is possible that *sibeliaZ* is not able to finish all alignments and returns its

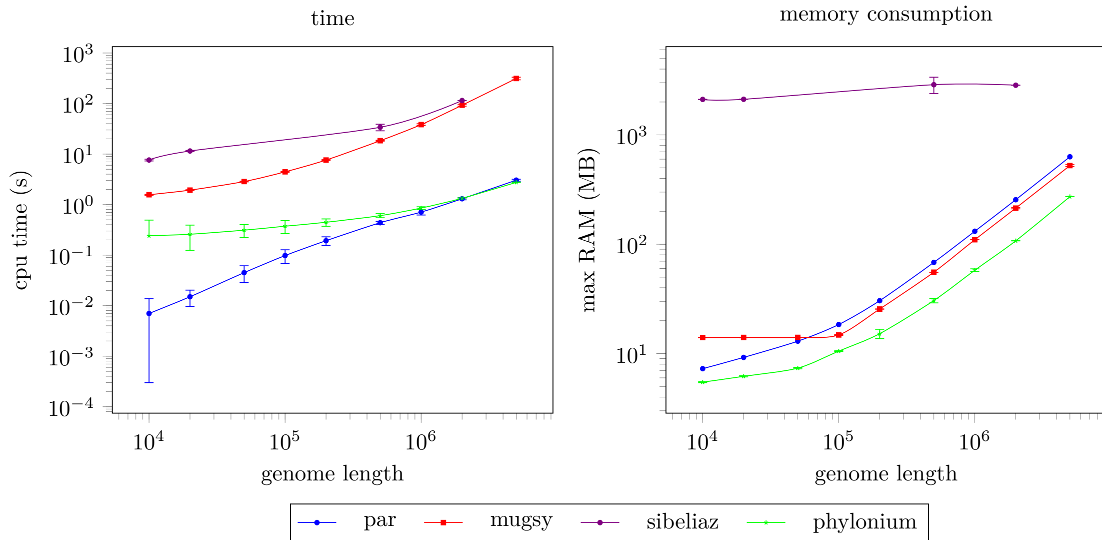


Figure 5.4: Time (left) and memory (right) consumption as a function of genome length.

current state, providing at least some alignments. *Par* terminates in under a second with an accuracy above 99.9%.

5.3 Real Data

Covid Data

For the 100 Covid Genomes *mugsy*, *par* and *sibeliaZ* produced valid alignments. *Par* terminated after 0.16 seconds and returned an alignment with an accuracy of 99.88% compared to *mugsy* which terminated after 90 seconds. *SibeliaZ* had almost identical accuracy (99.%) but took about 105 seconds to finish. For the 200 and 300 genomes both *mugsy* and *sibeliaZ* did not finish at all. *Par* terminated after 0.25 and 0.40 seconds respectively. *Par* was also able to align all 619 covid genomes, which took less than one second.

Eco29

On the Eco29 dataset all tools finished their alignments. We ran *mugsy* and *sibeliaZ* in default mode and *par* twice for each of the 29 genomes. The first run was with settings to filter for genomes with overlapping multiple homologous segments within the same genome and the second one allowing a sequence to align multiple times at the same positions on the reference (see Section 3.4). Running *par* for every genome also allows to evaluate the choice of the reference since this might change the outcome as well. See Table 5.2 for an

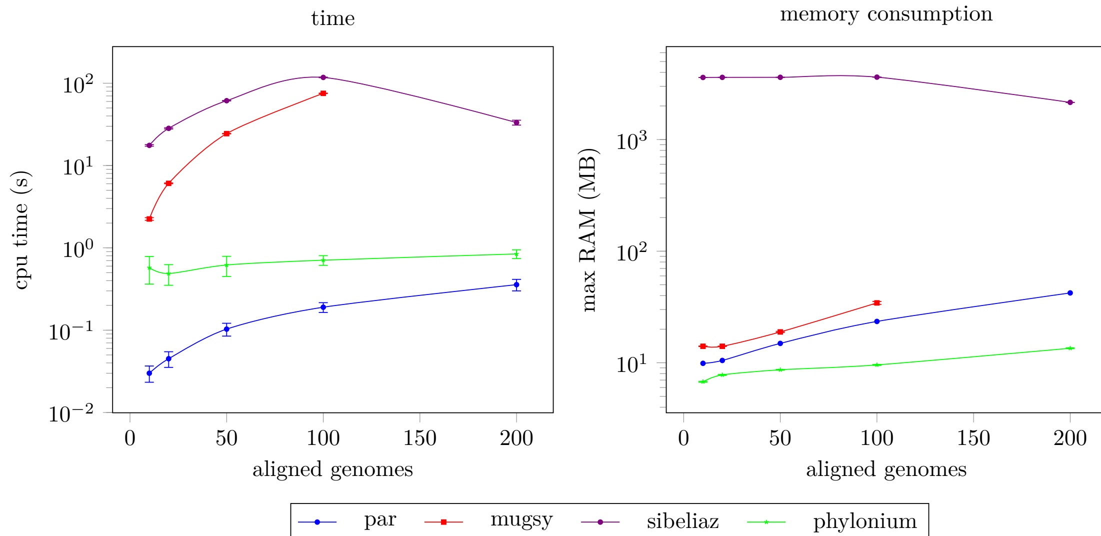


Figure 5.5: Time (left) and memory (right) consumption as a function of number of genomes compared.

dataset	tool	F-score	cpu time (s)	memory (MB)	elapsed clock time (s)
n100	mugsy	1.0000	93.53	161	91.06
n100	par	0.9988	0.20	24	0.16
n100	sibeliaZ	0.9987	121.70	593	105.51
n200	par	-	0.32	34	0.25
n300	par	-	0.50	51	0.40

Table 5.1: Performance on the covid data

overview over all results. We listed the results for the run of *par*'s choice of the reference seperately to the average results over al references. For both results, with and without filtering, the reference picked by par yields better results than other sequences on average. Also, both do not pick the reference that achieves the best possible results. In general, the results of *par* are less accurate compared to *sibeliaZ*. *Mugsy*'s alignments were considered as gold standard. Therefore it achieves 100%.

SibeliaZ and *mugsy* agree on 94% of the alignment, taking 1897 and 6905 seconds on the CPU, respectively. This means that *mugsy* took almost 2 hours on our system to finish the alignment. *Par* took on average around 11 seconds but at the cost of only 68% agreement for the reference it picked. This result gets even worse when the step for filtering homologies that overlap is skipped. The result for the default reference without filtering is at 60% and also on average, the results are worse without filtering. *Mugsy* and *SibeliaZ* both detect overlapping homologs, so an increase in agreement would be less surprising. The decline in F-score for *par* is mainly caused by a decline in precision (see

tool	F-score	Precision	Recall	cpu time (s)	memory (MB)	elapsed clock time (s)
mugsy	1.0000	1.0000	1.0000	6905	2884	6767
par (filter)	0.6876	0.7376	0.6439	10.32	663	04.87
parAll (filter)	0.6178	0.6771	0.5681	11	696	5.12
par (no filter)	0.6074	0.5632	0.6591	10.15	667	05.51
parAll (no filter)	0.5880	0.5981	0.5813	10	705	5.19
sibeliaZ	0.9487	0.9379	0.9597	1897	3467	148.08

Table 5.2: Performance on the Eco29 data. ParAll denotes the average results for each possible reference. Par depicts the result for the reference with median length picked by par.

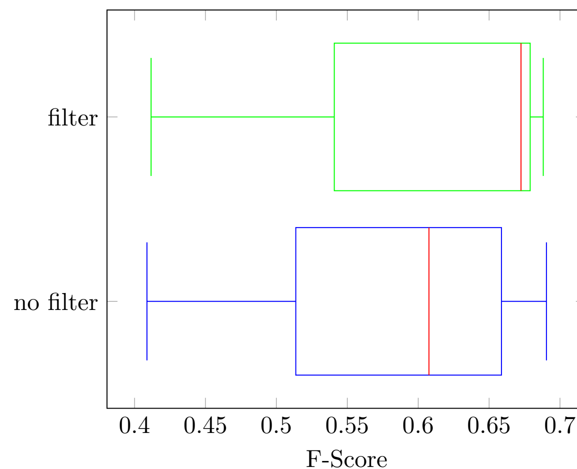


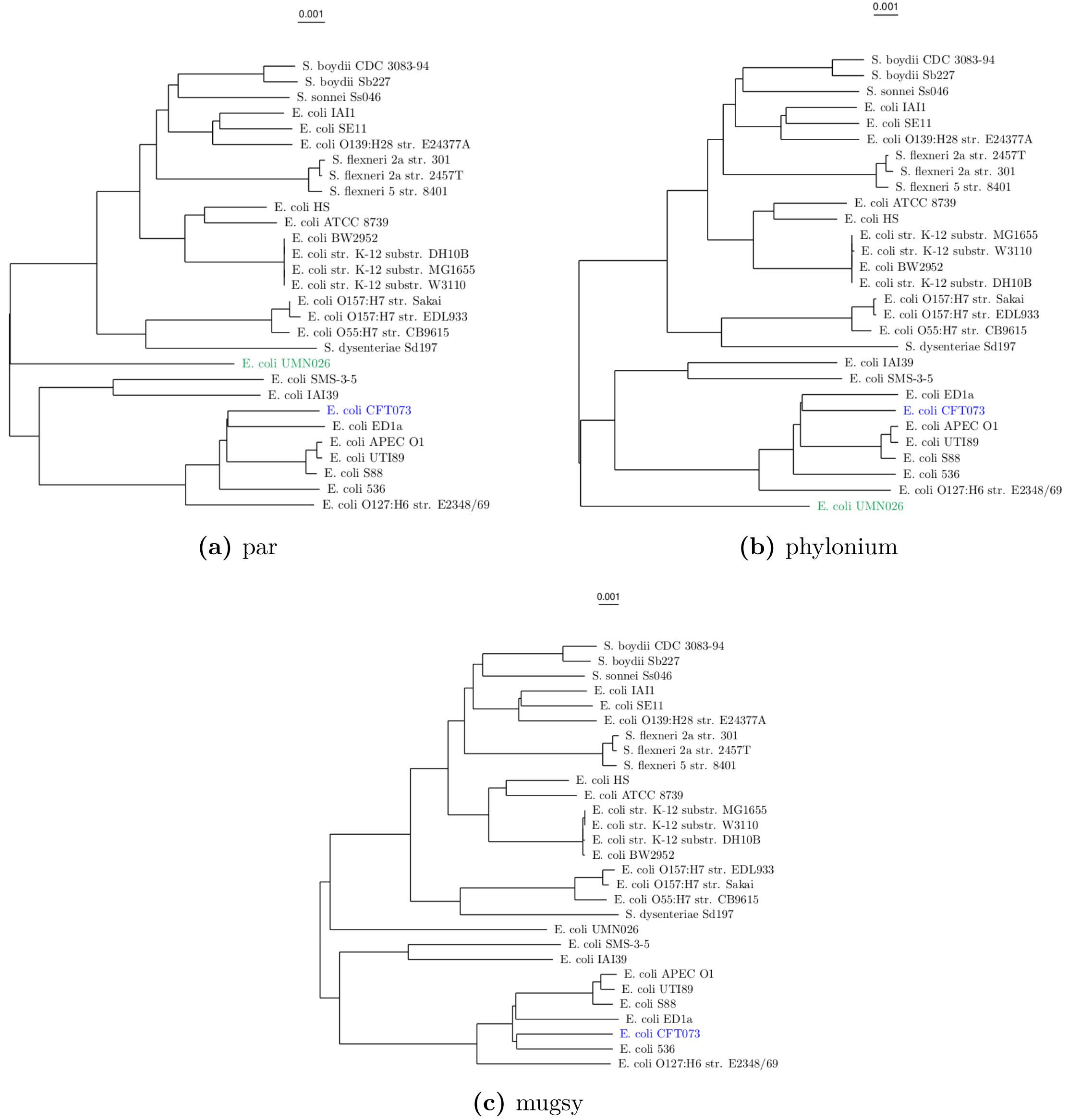
Figure 5.6: Distribution of results without (blue/bottom) and with filtering (green/-top) for overlapping homologies.

Table 5.2, column 3). The precision for the run without filtering drops by 8 points from 67% to 59%, whereas recall even increases by 1.5%. This means that *par* finds a few more correct alignments. In return, however, it also identifies many more false positive alignments. A possible explanation could be that without filtering, *par* finds too much or too short segments that are not considered by *mugsy*.

Figure 5.6 shows how the results for both *par* approaches are distributed. The best and the worst results for both approaches are similar. For many sequences though, the approach with filtering leads to better results than for the approach without filtering. For the filtering approach the median is much closer to the upper quantile. The approach without filtering leads to worse results in general but is also more evenly distributed which brings the median further down.

5.4 Comparison to Phylonium

As seen above, *par* and *phylonium* show similar behaviour in terms of runtime and memory consumption. Their alignment results are difficult to compare directly, though. *Phylonium* prints a distance matrix whereas the result of *par* is a multiple sequence alignment. But, both results can be used to compute a phylogeny and the resulting phylogenies can be compared instead. Figure 5.7 shows phylogenies generated from the results of *par*, *phylonium* and *mugsy*. The trees of *phylonium* and *par* show the same topology apart from a small detail, the position of *E. coli UMN026* annotated in green. Both trees are also similar to the tree based on the alignment by mugsy. The main difference of both trees is the clade at the bottom where strain of *E. coli CFT073* (blue) is located at an earlier branch by *mugsy*.



(a) par

(b) phylonium

(c) mugsy

Figure 5.7: Phylogenies produced by *par* (a) and *phylonium* (b) and *mugsy* (c).

6 Discussion

Increasing availability of large amounts of homologous sequence data provides a high demand for tools that are able to process them. Multiple sequence alignments can reveal information about the compared sequences such as their function or evolutionary distances between the genomes. The ability to create alignments for thousands of large and closely related genomes is a challenge that alignment-based methods are currently not able to solve (Armstrong et al., 2019). Alignment-free sequence methods are fast and accurate and offer a solution to address the challenge of comparing large sequences. However, they have the disadvantage that their results are difficult to attribute to specific segments on the genome. For most alignment-free methods, information on sequence localization is lost (Vinga, 2014).

With *phylonium*, Klötzl and Haubold (2019) presented a fast and accurate *alignment-free* tool based on micro-alignments. The micro-alignments were then taken to estimate the pairwise distances between the sequences. The purpose of this thesis was to reimplement the process of generating such micro-alignments and to construct a common multiple sequence alignment from them. The result is the tool *par* (Pile Anchors on Reference), which produces a multiple sequence alignment based on anchor matches.

In Chapter 5, the quality of these alignments was compared to the established multiple sequence alignment methods *mugsy* and *sibeliaZ*. For closely related sequences *par* produces accurate alignments that are quite competitive with those generated by *mugsy* and *sibeliaZ*. I illustrated this using simulated data sets. But also on empirical data like the covid dataset it was possible to obtain par results that agreed with *mugsy*. For increasing evolutionary distances, the accuracy of *par* starts to drop. Likewise, *mugsy* has already been criticized for a decline in accuracy on more distantly related genomes (Earl et al., 2014). It is a known issue that finding homologies is simpler with less evolutionary distance (Armstrong et al., 2019). For par, however, the accuracy drops even earlier compared to *mugsy* and on the E. coli genomes compared to *sibeliaZ* as well. The alignments on the E. coli data are only at around 68% accuracy, which is not adequate. In contrast, the trees on the E. coli dataset produced by *par*, like those produced with *phylonium*, are still accurate when compared to *mugsy*. This suggests that the alignments found using anchor matches are still sufficient to provide useful statistical information about the sequences. The estimated substitution rates on that basis are still good. Yet, for the evolutionary distance of the E.coli genomes, the anchor matches in *par* are less sufficient for constructing accurate explicit alignments.

On a more positive note, *par* uses less time and memory than the competition. This makes it possible to generate results in the first place when other tools fail, for example at aligning more than 100 covid genomes on a personal computer. Thus, a use for *par* could be on

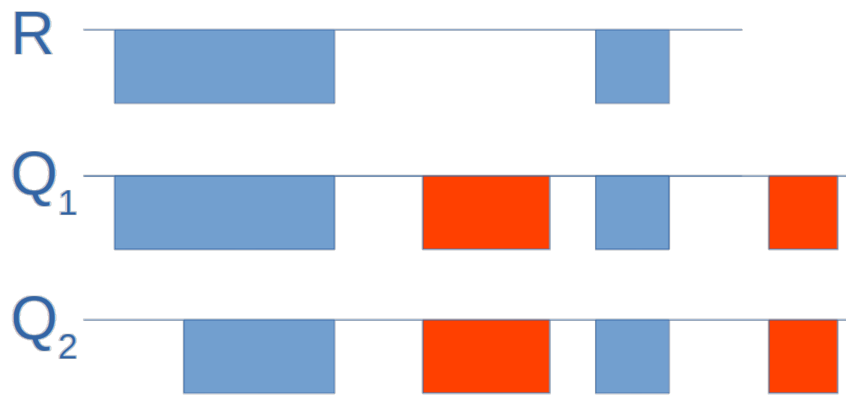


Figure 6.1: Reference biased alignment. The blue matches that are also located on the reference are found whereas the red matches are not detected.

large datasets of closely related genomes, e.g. as generated during pathogenic outbreaks. It could be helpful when regular alignment-based methods fail, whether the underlying cause is that they take too much time or the hardware requirements are too demanding. Therefore, the area of application overlaps with that of alignment-free methods such as *phylonium*. As both methods are based on anchor matches to a single reference, this is not surprising. In contrast to other alignment-free methods, it is possible to construct actual alignments using the anchor matches. This could be of use for a larger range of tasks in sequence analysis than mere distance computation, including the bootstrapping of phylogenies. Bootstrapping is a common approach to generating support values for individual nodes in phylogenies.

Possible Improvements

In *par* only a single sequence, the reference, is indexed and all other sequences are compared to it. This is a reason for the speed of *par*, but relying on a single reference effects accuracy at a certain point. As in any reference-based alignment method it is difficult to find matches that are not located on the reference. Matches that are only located on the query sequences are not found, since the queries are only compared with the reference. Figure 6.1 displays an example for this case. Here, the blue regions are detected and aligned since they occur in the reference sequence R as well. The regions in red could be compared between Q_1 and Q_2 but they are lost from the analysis because they are absent in R.

A way of improving the accuracy of *par* could be to examine the sequences further to find additional alignments. For example, long segments that are not found to contain any matches on the reference could be indexed and compared to other sequences. This would be at the expense of speed, but even compared to fast alignment tools like *mugsy*, *par* could afford to trade runtime for accuracy and still finish first. Also, an attempt could

be made to extend the matches on the other sequences after they cannot be continued on the reference. Additionally, the possibility to choose more than one sequence for indexing could be added. This would be similar to *andi* , where every sequence is indexed. *Andi* is not as fast as *phylonium* but still faster than *mugsy*. Here, a hybrid way to index only a part of the sequences and not every sequence could be a sensible solution. The new and faster calculation of the suffix array using *libsais* could be helpful here as well, since the repeated calculation of the suffix array is a bottleneck.

Finally, in order to investigate the use of anchor matches, they could be used as seeds for other alignment-based methods. One approach to this could be similar to Leimeister et al. (2018). They used *filtered spaced word matches* to generate seeds for *mugsy*. Similarly, the anchor matches could be used as seeds as well, since they provide good alignments already.

Bibliography

MI Abouelhoda, S Kurtz, and E Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi: https://doi.org/10.1016/S1570-8667(03)00065-0.

SF Altschul, W Gish, W Miller, EW Myers, and DJ Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. doi: https://doi.org/10.1016/S0022-2836(05)80360-2.

SV Angiuoli and SL Salzberg. Mugsy: fast multiple alignment of closely related whole genomes. *Bioinformatics*, 27(3):334–342, 12 2010. doi: 10.1093/bioinformatics/btq665.

J Armstrong, IT. Fiddes, M Diekhans, and B Paten. Whole-genome alignment and comparative annotation. In *ANNUAL REVIEW OF ANIMAL BIOSCIENCES, VOL 7*, volume 7 of *Annual Review of Animal Biosciences*, pages 41–64. 2019. doi: 10.1146/annurev-animal-020518-115005.

J Armstrong, G Hickey, M Diekhans, I Fiddes, AM Novak, A Deran, Q Fang, D Xie, S Feng, J Stiller, D Genereux, J Johnson, VD Marinescu, J Alfoldi, RS Harris, K Lindblad-Toh, D Haussler, E Karlsson, ED Jarvis, G Zhang, and B Paten. Progressive cactus is a multiple-genome aligner for the thousand-genome era. *NATURE*, 587(7833):246+, NOV 12 2020. doi: 10.1038/s41586-020-2871-y.

S Batzoglou. The many faces of sequence alignment. *Briefings in Bioinformatics*, Volume 6, Issue 1(Pages 6–22), 2005. doi: https://doi.org/10.1093/bib/6.1.6.

RS Boyer and JS Moore. A fast string searching algorithm. *Commun. ACM*, 20(10): 762–772, oct 1977. doi: 10.1145/359842.359859.

Magis C Chang JM Kemena C Bussotti G Erb I Notredame C Chatzou, M. Multiple sequence alignment modeling: methods and applications. *Briefings in bioinformatics*, 17(6)(1009–1023), 2016. doi: https://doi.org/10.1093/bib/bbv099.

A Darling, B Mau, F Blattner, and N Perna. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome research*, 14:1394–403, 08 2004. doi: 10.1101/gr.2289704.

AE Darling, B Mau, and NT Perna. progressivemauve: Multiple genome alignment with gene gain, loss and rearrangement. *PLOS ONE*, 5(6):1–17, 06 2010. doi: 10.1371/journal.pone.0011147.

CN. Dewey. *Whole-Genome Alignment*, pages 121–147. Springer New York, New York, NY, 2019. doi: 10.1007/978-1-4939-9074-0_4.

J Dhaliwal, SJ Puglisi, and A Turpin. Trends in suffix sorting: A survey of low memory algorithms. In *Proceedings of the Thirty-Fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, page 91–98, AUS, 2012. Australian Computer Society, Inc. ISBN 9781921770036.

JY Dutheil, S Gaillard, and E Stukenbrock. Maffilter: a highly flexible and extensible multiple genome alignment files processor. *BMC Genomics*, 15, 2014. doi: 10.1186/1471-2164-15-53.

D Earl, N Nguyen, G Hickey, RS Harris, S Fitzgerald, K Beal, I Seledtsov, V Molodtsov, BJ Raney, H Clawson, J Kim, C Kemena, JM Chang, I Erb, A Poliakov, and M et. al Hou. Alignathon: a competitive assessment of whole-genome alignment methods. *GENOME RESEARCH*, 24(12):2077–2089, DEC 2014. doi: 10.1101/gr.174920.114.

J Felsenstein. *Inferring Phylogenies*. Sinauer, 2004. ISBN 9780878931774.

DF Feng and RF Doolittle. Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

J Fischer and F Kurpicz. Dismantling divsufsort, 2017.

MC Frith and AMS Shrestha. A simplified description of child tables for sequence similarity search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(6):2067–2073, 2018. doi: 10.1109/TCBB.2018.2796064.

AE Gorbalenya, SC Baker, RS Baric, RJ de Groot, C Drosten, AA Gulyaeva, BL. Haagmans, C Lauber, AM Leontovich, BW Neuman, D Penzar, S Perlman, LLM Poon, DV Samborskiy, IA. Sidorov, I Sola, J Ziebuhr, and Coronaviridae Study Grp. The species severe acute respiratory syndrome-related coronavirus: classifying 2019-ncov and naming it sars-cov-2. *Nature Microbiology*, 5(4):536–544, APR 2020. doi: 10.1038/s41564-020-0695-z.

D Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574931.

B Haubold. Alignment-free phylogenetics and population genetics. *Briefings in Bioinformatics*, 15(3):407–418, 11 2013. doi: 10.1093/bib/bbt083.

B Haubold and T Wiehe. Comparative genomics: methods and applications. *Naturwissenschaften*, 91(9):405–421, SEP 2004. doi: 10.1007/s00114-004-0542-8.

B Haubold and T Wiehe. *Introduction to Computational Biology: An Evolutionary Approach*. Birkhäuser Basel, 2006. ISBN 978-3-7643-6700-8. doi: https://doi.org/10.1007/3-7643-7387-3.

B Haubold, P Pfaffelhuber, M Domazet-Los˘o, and T Wiehe. Estimating mutation distances from unaligned genomes. *Journal of Computational Biology*, 16(10):1487–1500, 2009. doi: 10.1089/cmb.2009.0106.

B Haubold, F Klötzl, and P Pfaffelhuber. andi: Fast and accurate estimation of evolutionary distances between closely related genomes. *Bioinformatics*, 31(8):1169–1175, 12 2014. doi: 10.1093/bioinformatics/btu815.

RR Hudson. Generating samples under a Wright–Fisher neutral model of genetic variation . *Bioinformatics*, 18(2):337–338, 02 2002. doi: 10.1093/bioinformatics/18.2.337.

S Iantorno, K Gori, N Goldman, M Gil, and C Dessimoz. *Who Watches the Watchmen? An Appraisal of Benchmarks for Multiple Sequence Alignment*, pages 59–73. Humana Press, Totowa, NJ, 2014. ISBN 978-1-62703-646-7. doi: 10.1007/978-1-62703-646-7_4. URL https://doi.org/10.1007/978-1-62703-646-7_4.

NC Jones and PA Pevzner. *An Introduction to Bioinformatics Algorithms.* MIT Press, Cambridge MA, 2004.

TH Jukes and CR Cantor. Evolution of protein molecules. In *Mammalian Protein Metabolism*, pages 21–132. Academic Press, 1969.

J Kärkkäinen and Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, 2003.

RM Karp, RE Miller, and AL Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, page 125–136, New York, NY, USA, 1972. Association for Computing Machinery. ISBN 9781450374576. doi: 10.1145/800152.804905.

T Kasai, G Lee, H Arimura, A Setsuo, and K Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. volume 2089, pages 181–192, 06 2001. doi: 10.1007/3-540-48194-X_17.

K Katoh. *Multiple Sequence Alignment Methods and Protocols: Methods and Protocols.* 01 2021. ISBN 978-1-0716-1035-0. doi: 10.1007/978-1-0716-1036-7.

K Katoh, K Misawa, K Kuma, and T Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, 30(14): 3059–3066, 07 2002. doi: 10.1093/nar/gkf436.

B Kehr, K Trappe, M Holtgrewe, and R Knut. Genome alignment with graph data structures: A comparison. *BMC bioinformatics*, 15:99, 04 2014. doi: 10.1186/1471-2105-15-99.

DS Kim, JS Sim, H Park, and K Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 186–199. Springer Berlin Heidelberg, 2003. doi: https://doi.org/10.1007/3-540-44888-8_14.

F Klötzl. *Fast Computation of Genome Distances.* PhD thesis, University of Lübeck, Oct 2020.

F Klötzl and B Haubold. Phylonium: fast estimation of evolutionary distances from large samples of similar genomes. *Bioinformatics*, 36(7):2040–2046, 12 2019. doi: 10.1093/bioinformatics/btz903.

DE Knuth. *Literate Programming*. The Center for the Study of Language and Information Publications, 1992.

DE Knuth, Morris JH, and VR Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi: 10.1137/0206024.

P Ko and S Aluru. Space efficient linear time construction of suffix arrays. volume 3, pages 200–210, 01 2003. doi: 10.1016/j.jda.2004.08.002.

CA Leimeister, T Dencker, and B Morgenstern. Accurate multiple alignment of distantly related genome sequences using filtered spaced word matches as anchor points. *Bioinformatics*, 35(2):211–218, 07 2018. doi: 10.1093/bioinformatics/bty592.

U Manber and G Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi: 10.1137/0222058.

I Minkin and P Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with sibeliaz. *NATURE COMMUNICATIONS*, 11(1), DEC 10 2020. doi: 10.1038/s41467-020-19777-8.

B Morgenstern. Sequence comparison without alignment: The spam approaches. In K Katoh, editor, *Multiple Sequence Alignment: Methods and Protocols*, volume 2231 of *Methods in Molecular Biology*, pages 121–134. 2021. doi: 10.1007/978-1-0716-1036-7_8.

SB Needleman and CD Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. doi: https://doi.org/10.1016/0022-2836(70)90057-4.

Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. doi: 10.1109/TC.2010.188.

E Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

B Paten, J Herrero, K Beal, S Fitzgerald, and E Birney. Enredo and pecan: Genome-wide mammalian consistency-based multiple alignment with paralogs. *Genome research*, 18: 1814–28, 11 2008. doi: 10.1101/gr.076554.108.

SJ. Puglisi, WF Smyth, and AH Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), jul 2007. doi: 10.1145/1242471.1242472.

N Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994. doi: 10.1109/52.311070.

N Saitou and M Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.

A Shrestha, M Frith, and P Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in bioinformatics*, 15, 01 2014. doi: 10.1093/bib/bbt081.

TF Smith and MS. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. doi: 10.1016/0022-2836(81)90087-5.

JD Thompson, DG Higgins, and TJ Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 11 1994. doi: 10.1093/nar/22.22.4673.

N Timoshevskaya and W Feng. Sais-opt: On the characterization and optimization of the sa-is algorithm for suffix array construction. In *2014 IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–6, 2014. doi: 10.1109/ICCABS.2014.6863917.

R Van Noorden, B Maher, and R Nuzzo. The top 100 papers. *Nature News*, 514(7524): 550, 2014.

S Vinga. Editorial: Alignment-free methods in computational biology. *Briefings in Bioinformatics*, 15(3):341–342, 05 2014. doi: 10.1093/bib/bbu005.

S Vinga and J Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 03 2003. doi: 10.1093/bioinformatics/btg005.

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

P Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, 1973. doi: 10.1109/SWAT.1973.13.

Jing Yi Xie, Ge Nong, Bin Lao, and Wentao Xu. Scalable suffix sorting on a multicore machine. *IEEE Transactions on Computers*, 69(9):1364–1375, 2020. doi: 10.1109/TC.2020.2972546.

Y Zhang, Q Zhang, J Zhou, and Q Zou. A survey on the algorithm and development of multiple sequence alignment. *Briefings in Bioinformatics*, 23(3), 03 2022. doi: 10.1093/bib/bbac069.

A Zielezinski, S Vinga, J Almeida, and W Karlowski. Alignment-free sequence comparison: Benefits, applications, and tools. *Genome Biology*, 18:186, 10 2017. doi: 10.1186/s13059-017-1319-7.

Girgis HZ Bernard G et al. Zielezinski, A. Benchmarking of alignment-free sequence comparison methods. *Genome Biology 20*, 20:144, 2019.