UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
NEURO- UND BIOINFORMATIK

Efficient Estimation of Evolutionary Distances

Effizientes Schätzen von evolutionären Distanzen

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

Vorgelegt von
Fabian Klötzl

Ausgegeben und betreut von
Prof. Dr. Bernhard Haubold

Lübeck, den 1. April 2015 (last modified on 2015-10-21)

Version Notice

This version of my Master's thesis contains a few changes over the official version. I removed the copyright-protected Tree of Life picture in the bio chapter. Likewise I removed the declaration of independence or whatever its called. Finally, this notice and the license were added.
— Lübeck, 2015-10-21; Fabian Klötzl

License

Acknowledgments

I would like to thank the Institute for Theoretical Computer Science of the University of Lübeck for letting me run tests on their 64 core machine.
Special thanks go to Bernhard Haubold and the Max Planck Institute for Evolutionary Biology in Plön for supporting me with my work.

Abstract

The advent of *high throughput sequencers* has lead to a dramatic increase in the size of available genomic data. Standard methods, which have worked well for many years, are not suitable for the analysis of big data sets, due to their reliance on a time-consuming *alignment* step. In this thesis, a new *alignment-free* approach for *phylogeny reconstruction* is introduced. The corresponding program, andi, is orders of magnitude faster than classical approaches and also superior to comparable *alignment-free* methods.

The central data structure in andi is the *enhanced suffix array*. It is used to find long exact matches between sequences. In this thesis, various approaches to the construction of enhanced suffix arrays, including novel ones, are evaluated with respect to performance. Additionally, a new parallel algorithm for the computation of *suffix arrays* is introduced.

Zusammenfassung

Mit der Einführung von *Next-Generation*-Sequenzierer-Techniken ist die verfügbare Menge von Genomdaten erheblich gewachsen. Standardansätze, wie das *Alignment*, die über Jahre hinweg gut funktionierten, kommen bei großen Datenmengen an ihre Grenzen. In dieser Arbeit wird ein neuer, *alignment-freier* Ansatz zur *Phylogenierekonstruktion* vorgestellt. Dessen Implementierung, andi, ist um Größenordnungen schneller als klassische Methoden und auch vergleichbaren *alignment-freien* Programmen überlegen.

Die zentrale Datenstruktur in andi ist das sogennante *Enhanced Suffix Array* (ESA). Es wird dazu benutzt, lange exakte Übereinstimmungen zwischen Sequenzen zu finden. Um diesen Prozess schnellstmöglich zugestalten, werden in dieser Arbeit verschiedene Konstruktionsansätze für ESAs evaluiert. Dazu gehört auch ein neuer, paralleler Algorithmus zur Berechnung von *Suffix Arrays*.

Contents

1 Introduction

Creating a phylogeny is a standard step done early in analysis of related genomes. With it, the genomes are clustered, outliers can be detected, and further analyses can be planned. Thus, phylogenies are an integral part of most multi-genome studies.

Given a set of genomic sequences, a phylogeny can be computed via different approaches: *maximum parsimony* and *maximum likelihood* search for the tree that best explains the data, where *best* is some criteria based on the method. However, for n genomes, the number of possible rooted phylogenetic trees is $C_n = 1/(n+1)\binom{2n}{n}$ the nth Catalan number. C_n can be approximated as $C_n \approx 4^n\, n^{-3/2}\, \pi^{-1/2}$. Thus, the Catalan numbers and the number of possible phylogenies, grow nearly exponentially.

A third approach is based on a matrix of pairwise distances. From this matrix, a phylogeny can be generated using standard algorithms, such as *neighbor joining*. So for this approach, only a single tree is computed, based on n^2 evolutionary distances. These distances are traditionally computed via an alignment (see Section 2.2). However, alignments are slow and thus, unfeasible, if n is large: For the biggest data set in this thesis, consisting of 3085 *S. pneumoniae* genomes, an alignment would take 3.2 years. Thus, in recent years, *alignment-free* methods have been developed, which also estimate evolutionary distances, but are much faster than an alignment.

In this thesis, a new method, called *anchor distance* or simply andi, for the estimation of evolutionary distances is developed. Simulations show that andi is accurate for closely related sequences, even when combined with high levels of recombination. Applications to real data show that andi is more accurate than existing estimation methods.

Furthermore, through careful engineering, andi has supreme performance. When applied to big datasets, it is orders of magnitude faster than the classic *alignment*, and still significantly faster than all other *alignment-free* methods evaluated in this thesis. For example, on the same data set of 3085 genomes, andi takes only a few hours.

The central data structure of andi is the *suffix array*. This array includes the indices to all suffixes of a text in lexicographic order. Various so-called *suffix array construction algorithms* have been developed in the past 15 years. In this thesis, a new algorithm is introduced, targeted at the *parallel* construction of a suffix array.

Chapter 2 goes into further detail in the computation of phylogenies. It includes short descriptions on previous approaches for the estimation of evolutionary distances. These distances are then evaluated, with respect to their accuracy and performance on simulated data.

To set the stage for an introduction to andi, all the necessary algorithms and data structures are explained in Chapter 3. This includes text-book approaches, like the *enhanced suffix array*, as well as new ideas, such as the *first variant character*. The algorithms for the creation of suffix arrays have been separated into Chapter 4. In that chapter, a new parallel algorithm is introduced, its correctness proven, and its complexity analyzed.

The anchor distance for the estimation of evolutionary distances is defined in Chapter 5. This method is analyzed for its computational complexity, and worst-case accuracy. Also

the pseudocode, as well as hints, for an efficient implementation, are given.

In Chapter 6, both andi, and the new suffix array construction algorithm, are evaluated for precision and efficiency. andi and other distance estimators are applied to various simulated test, to measure their accuracy. For performance evaluations, all methods are applied to real sets of bacterial genomes. Similarly, the new algorithm is tested on engineered corpi, as well as common text inputs.

Chapter 7 concludes this thesis with an analysis of the results. It also includes suggestions for improvements of both andi and the new algorithm. For andi, ideas for better accuracy and improved performance are proposed.

2 Biological Background

2.1 Evolutionary Distances and Phylogenies

There are approximately 1.9 million described and 11 million undescribed species in the world [Chapman, 2009]. These numbers are only a rough estimate with new species continuously being born and going extinct. Efforts to bring order into these vast numbers date back to Linné 1758 [Campbell and Reece, 2011] and thus, predate Darwin's theory of evolution [Darwin, 1859] and even more so modern genetics [Morgan et al., 1915]. In the absence of evolutionary data, Linné build the now classical taxonomy of all living things on morphological features with the smallest unit being a species.

> »I can entertain no doubt, [...] that the view which most naturalists entertain, and which I formerly entertained—namely, that each species has been independently created—is erroneous. I am fully convinced that species are not immutable; but that those belonging to what are called the same genera are lineal descendants of some other and generally extinct species, in the same manner as the acknowledged varieties of any one species are the descendants of that species.«
> — Charles Darwin, *The Origin of Species*; *p. 61*

In the above quote from the introduction of *The Origin of Species* Darwin expresses the idea that species, classified in a common genus because of shared morphological features, are descendants from a single ancestral species. So at one point in time there was a species with certain features and over time its descendants gradually differentiated into the diverse taxa that can be observed today. It is this claim which shows that the taxonomy by Linné, which is based on shared morphology, represents evolution.[1]

Figure 2.1 depicts a beautiful tree of life. It shows, for example, that the last common ancestor of mammals and reptiles lived about 250 millions years ago and that birds diverged 110 million years ago. So today's reptiles are more closely related to birds than to mammals. This relatedness is called *evolutionary distance*.

Intuitively, evolutionary distance is measured in time, as seen above. However, the unit depends on the available data: millions of years for fossils, generations in experiments with microbes, and mutation rates in genomics. In the latter case, the *molecular clock* is used to translate substitution rates to years [Zuckerkandl and Pauling, 1962].

Throughout this thesis an *evolutionary distance* of genomes is defined as a real number from the interval $[0, \infty)$. A distance of $d = 0$ means that two sequences are identical whereas a distance $d \geq 1$ means that they are presumably unrelated. This is especially true for simulated sequences without a common ancestor.

[1] Since the concept of evolution was unknown to Linné his taxonomy resembles the contemporary belief in the fixity of species. This becomes apparent in that the lower levels of his taxonomy closely relate to an evolutionary history, whereas the higher levels are rather artificial [Campbell and Reece, 2011].

Here be a *Tree of Life* figure. For the original go to `https://www.evogeneao.com/learn/tree-of-life`.

Figure 2.1: A representation of the evolutionary relationships between all extant organisms and a few extinct groups. Notice the dashed lines that indicate time points in the past.
Printed with permission.

$$D = \begin{pmatrix} 0 & 0.1 & 0.25 & 0.3 \\ 0.1 & 0 & 0.3 & 0.3 \\ 0.25 & 0.3 & 0 & 0.05 \\ 0.3 & 0.3 & 0.05 & 0 \end{pmatrix}$$

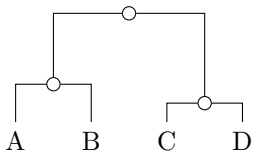


Figure 2.2: A simple phylogeny of four imaginary species $M = \{A, B, C, D\}$. The sequences C and D are closer related than the pair A and B. Each pair forms its own clade in the phylogenetic tree. The tree (a *rooted phylogram*) was computed from the distance matrix using the UPGMA algorithm as implemented in the Ti*k*Z phylogeny drawing package [Mäusle, 2012].

Let M be a set of genome sequences. Then a function $d\colon M \times M \to [0, \infty)$ represents the evolutionary distances on M if it is a metric.[2] From this a matrix $D_{ij} = d(i, j)$ can be derived. This distance matrix is symmetric and its main diagonal contains only zeros.

Given these distances—either via d or implicitly via D—the underlying phylogenetic tree can be reconstructed [Felsenstein, 2004]. It represents the evolution of the given organisms with each internal node being the common ancestor of all its subspecies. The species from set M are the leaves of the tree. An example matrix and its corresponding tree is given in Figure 2.2.

2.2 Estimating Evolutionary Distances

In the previous section we established that the phylogeny of a group of organisms can be reconstructed via their evolutionary distances. In this section various approaches are explained to estimate distances from genome sequences [Haubold, 2014], before introducing the new method I codeveloped in Chapter 5.

Mutation Rate

One of the essential forces behind evolution is mutation. On the level of deoxyribonucleic acid (DNA) the simplest mutation is the substitution of one nucleotide with a different one (e. g., Adenine to Thymine, $\mathsf{A} \to \mathsf{T}$). Assuming that all species are subject to the same mutation rate, their mutual single nucleotide polymorphism (SNP) rate may be a good estimator for the evolutionary distance [Zuckerkandl and Pauling, 1962].

Let Q and S be the sequences TTAAGTAAGG and TTACGTCAGG, respectively. Then the *Hamming distance* is defined as the ratio of mismatches, $d_H(S, Q) = 0.2$. But this only accounts for the *observed* substitutions. Given enough time the nucleotide at a certain position may mutate multiple times, resulting in a neutral mutation (e. g., $\mathsf{A} \to \mathsf{T} \to \mathsf{A}$). Such an invisible mutation accounts for two or possibly more substitutions. The simplest model to correct for these is known as the *Jukes-Cantor correction* [Jukes and Cantor, 1969].

Definition 1. *Let d be an evolutionary distance. Then the* Jukes-Cantor correction *is*

$$JC(d) = -\frac{3}{4} \ln\left(1 - \frac{4}{3}d\right) .$$

[2] Some models use evolutionary distances which are *ultra metric* e. g., [Daskalakis and Roch, 2013] These models lead to equal branch lengths for all leaves but are rarely used in practice.

S: A T T C G T
Q: A – T C C T

Figure 2.3: The table shows one possible alignment for the sequences ATTCGT and ATCCT. A gap is denoted by »–«. Another possible alignment would have first the T at position 2 in Q, followed by the gap. All other alignments would require more than two edits (one gap and one mismatch).

Unfortunately, substitutions are not the only mutations; but indels,[3] unequal crossing over and inversions move chunks of genomic sequences along genomes. This renders the simple Hamming distance useless, unless applied only to homologous sequences. The *anchor distance* strategy for finding pairs of homologous sequences is presented in Chapter 5.

Alignment

To overcome the limitation of the *Hamming distance* with respect to indels, the *Levenshtein distance* d_L (also known as *edit distance*) is used instead. It is defined as the smallest number of insertions, deletions or substitutions one has to do in order to transform one sequence into the other. This can be visualized by *aligning* the two sequences in question (see Figure 2.3). The value d_L is then called the *score* of the alignment.

The *optimal* alignment of two sequences (i. e., the alignment with fewest edits) with lengths n and m, can be computed in $O(nm)$ time and $O(\min\{n, m\})$ space, using dynamic programming [Ohlebusch, 2013, p. 397]. Heuristic methods are known that compute approximate alignments in expected linear time [Altschul et al., 1990]. Optimally aligning more than two sequences (i. e., a *multiple sequence alignment*), however, has been proven to be NP-complete [Wang and Jiang, 1994].

Interestingly, when calculating the evolutionary distance from an alignment, indels are disregarded [Felsenstein, 2004]. Instead, only the relative amount of mismatches is calculated. Thus, if an *alignment-free* method can avoid the computation of indels, it may estimate of the evolutionary distance much faster in practice.

Exact Word Count

A consecutive sequence of k nucleotides is known as a k-mer or k-tuple. Computing the frequency profile of all k-mers allows for easy comparison of two sequences. Let q_i be the frequency of pattern i in sequence Q then a simple distance definition is

$$d_{kmer}(Q, S) = \sum_{i=1}^{4^k} (q_i - s_i)^2 \,. \tag{2.1}$$

This method can be implemented efficiently [Marçais and Kingsford, 2011]. Unfortunately, it lacks accuracy when applied to closely related genomes [Haubold, 2014]. Efforts have been made to correct this defect, but they still either lack power or they have no freely available reference implementation [Maurer-Stroh et al., 2013]. Further, the choice for the best k remains unknown [Tang et al., 2014].

[3] As it is often impossible to distinguish between insertions and deletions in pairwise alignments, they are commonly referred to as *indels*.

Inexact Word Count

A number of strategies use *patterns* rather than words. These patterns, also called *spaced words* or *structures* are approximate matches. Consider the sequence ACCGCTG; then the 5-mer ATCGC is not contained, but the pattern AxCxC matches at position[4] 0, where x denotes a *do-not-care* or *wild-card* position.

Recently, [Morgenstern et al., 2014] devised a generalization of the exact word count. They use a bit pattern to reduce a k-mer to a l-mer where the l positions marked with a 1 are picked from the k-mer. The implementation by [Leimeister et al., 2014] uses multiple patterns to compute more accurate distances.

Definition 2. *For a nucleotide α the relative frequency is f_α and further $f := f_A^2 + f_C^2 + f_G^2 + f_T^2$. The* spaced word *distance is then defined as*

$$d_{sw}(Q,S) = -\frac{3}{4}\ln\left(\frac{4}{3}\sqrt[k]{\frac{N^{bin}(Q,S,\mathcal{P})}{|\mathcal{P}|\,(|Q|-l+1)} - 2\cdot(|S|-l)\cdot f^k} - \frac{1}{3}\right),$$

where $N^{bin}(Q,S,\mathcal{P})$ is the number of words matched by a pattern $P \in \mathcal{P}$ in both Q and S.

Unlike the previous distances, the method by [Yi and Jin, 2013] tries to find different matches across sequences rather than common patterns. Let P be a bit pattern e. g., 11011. Then ACxCT is a *context* on the sequence TTACGCTGA with the so-called *object* G being the sole nucleotide matching the do-not-care. On the other hand, TxA has two possible objects T and G. So TxA is not a valid context.

As usual, let Q, S be two sequences and C_Q, C_S be their set of contexts, respectively. Further $\delta\colon C_Q \times C_S \to \{0,1\}$ is 0 iff the given contexts share a common object. With $R = C_Q \cap C_S$ the *context-object distance* is defined as

$$d_{co}(Q,S) = \frac{\sum_{c\in R}\delta(c,c)}{|R|}. \tag{2.2}$$

Substitutions from Common Substrings

One of the most widely referenced alignment-free distance estimation methods is d_{kr} by [Haubold et al., 2009]. It is based on the following characteristic.

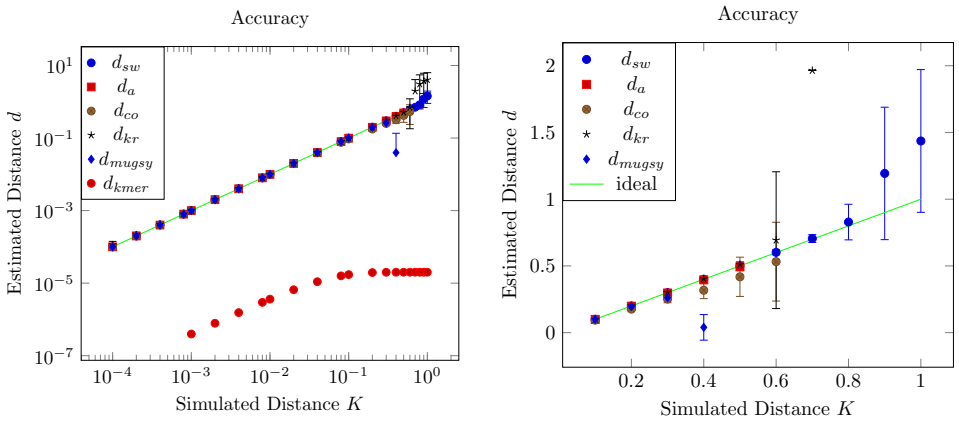
Definition 3. *Let S, Q be sequences over a common alphabet Σ. Then $m_S(Q^i)$ is the longest prefix of Q^i matching somewhere in S (compare Section 3.6). Further, the* matching statistics *of Q with respect to S is*

$$ms[i] = |m_S(Q^i)|.$$

A *glocal* alignment—local in S and global in Q^i—is assumed. Then $ms[i]$ is the distance from i to the next mutation.[5] The average distance to the next mutation is assumed to be approximately the inverse of the mutation rate, which is true under a uniform distribution of mutations.

[4] All indices in this thesis are zero-based.

[5] The implementation kr by [Domazet-Lošo and Haubold, 2009] uses *local shortest unique substrings* or *shustrings* which are $ms[i]+1$.



(a) log/log scale for 0.0001 to 1

(b) linear scale for 0.1 to 1

Figure 2.4: Estimation of the substitution rates for different distances. Shown are means and variance. An ideal estimator would have all its data points on the straight line. For the used implementations and parameters consult Table 2.1.

Definition 4. *Let ms be the matching statistics of Q with respect to S. Then the distance $d_{kr}(Q, S)$ is defined as*

$$d_{kr}(Q, S) = JC\left(\frac{1}{|Q|}\sum_i ms[i]\right). \tag{2.3}$$

2.3 Comparison of Prior Art

The previously described distance estimation methods vary widely in accuracy and performance. To motivate the need for better alignment-free methods we present a small comparison based on the distance estimation for two sequences with varying substitution rates.

Accuracy

Figure 2.4 shows measurements for the previously described distances. Each data point is the mean of one hundred runs. For each run a sequence pair of length $100\,\mathrm{kbp}$ is simulated with a substitution rate K (Jukes-Cantor corrected). An ideal distance estimation method would calculate the exact substitution rate of the input and hence, have all its data points on the straight line.

As can be seen in Subfigure 2.4a, the k-mer based estimation is monotone, but at least one order of magnitude smaller than expected. This lack of accuracy makes it inferior to all other methods.

The high number of substitutions makes good estimations for all methods increasingly difficult beyond a rate of 0.4. For a higher K value, d_{co} becomes downwards biased and stops working at 0.7. d_a fails beyond rates of $K \geq 0.5$. d_{kr} rapidly overestimates the distance for $K \geq 0.7$. For improved clarity its datapoints beyond $K = 0.7$ are omitted in Subfigure 2.4b. The best results for high substitution rates are produced by d_{sw}. Its

Table 2.1: Performance Comparison; sorted by runtimes.

Method	Implementation	Time (s)	Memory (KB)
d_{kr}	kr	292.45	9940
d_{co}	cophylog	399.57	156 852
d_a	andi	604.87	22 932
d_{kmer}, $k = 20$	jellyfish	673.66	3980
alignment	mugsy, dnadist	842.24	66 816
d_{sw}, $k = 20$	spaced	2595.59	4396

estimations are reliable up to $K = 0.8$. Beyond that, they become upwards biased and start fluctuating heavily.

As a reference, mugsy was used to compute alignments under the same conditions [Angiuoli and Salzberg, 2011]. From the alignment, the program dnadist from the Phylip toolbox was used to compute Jukes-Cantor corrected distances [Felsenstein, 2005]. Unsurprisingly, the alignment is among the most accurate estimations up to $K = 0.3$. For $K = 0.4$ its reported distances are one order of magnitude too small. For bigger K, no alignment is produced.

Performance

The fundamental reason for the invention of *alignment-free* distance estimation methods is their superior performance. Here performance has two characteristics: runtime and memory usage. The memory usage becomes more important for bigger data sets, because excessive memory usage may exceed the available memory and thus, limits a method's usability.

As a simple test, the runtime and maximum memory for the computation of all data points in Figure 2.4 were measured. Thus, for each of the 22 different K values, every method had to compute distances for 100 pairs of randomly generated $100\,\mathrm{kbp}$ sequences. The measurements taken by UNIX command time are presented in Table 2.1.[6] All implementations were run on a standard desktop computer (see Section 6.1; single-threaded; with default parameters, unless stated otherwise).

All alignment-free methods, except for spaced, were faster than the reference mugsy. This may be due to the chosen value for k, which is double the default $k = 10$, but produces more accurate results. As with d_{kmer}, the optimal value for k is unknown.

cophylog is the only method using more memory than mugsy. Manually changing some magic numbers in its code might result in a smaller hashmap and thus, reduced memory usage. However, a heuristic for the optimal hash size is missing.

[6] Some overhead due to input creation, formatting, and shell scripts may apply.

3 Algorithms and Data Structures

Looking for short patterns within long texts is a common problem in computer science. In fact, it arises so frequently that in *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein [Cormen et al., 2009] an entire chapter is devoted to *String Matching*. They formalize the *string-matching problem* as follows.

Definition 5 (String-Matching Problem)**.** *Let Σ be an alphabet of size $|\Sigma| = \sigma$. Further let u, v be words over Σ so that $u \in \Sigma^n$ and $v \in \Sigma^m$ with $m \leq n$. Then v is called a* substring *of u iff there exists an i such that $u[i..i+m-1] = u_i u_{i+1} \cdots u_{i+m-1} = v$.*

Now let u, v satisfying the above conditions be given. The string-matching problem *is then the task of*

a) checking if v is a substring of u,

b) finding all indices i for which the above is true.

In the past, various techniques for matching strings have been developed. Among the fastest is the well-known algorithm by Knuth, Morris, and Pratt. Its runtime is linear $\Theta(m+n)$ with the space requirement being just $O(1)$ [Cormen et al., 2009]. Less well-known algorithms have an expected sublinear runtime [Cantone and Faro, 2014].

DNA is a sequence of nucleotides, of which there are four kinds: Adenine, Cytosine, Guanine and Thymine.[1] So a DNA sequence can be considered a word over the alphabet $\Sigma = \{A, C, G, T\}$. If two DNA sequences are closely related, they share common subsequences, interrupted by mutations. Thus, if one can locate equal subsequences, the complement gives the mutations necessary for the calculation of evolutionary distances (see Section 2.2). So resorting to the string matching problem allows us to indirectly find mutations.

Let S be a subject DNA sequence and q be a short l-mer from a longer sequence Q ($|Q| = |S| = n$). Checking if q is a substring of S takes time $O(l+n)$ with the algorithm by Knuth, Morris, and Pratt. Since Q consists of n/l many l-mers, a full comparison would take $O(n/l \cdot (l+n))$ time.[2] But in each comparison S does not change, so we are interested in a comparison method with an asymptotic runtime of $O(n + n/l \cdot l) = O(n)$; that is, a procedure, which processes the subject and the query only a fixed number of times. In this chapter we establish the algorithms and data structures that achieve this goal at the cost of memory and increased preprocessing time.

3.1 Suffix Arrays

A suffix array (SA) of a text T contains all suffixes in lexicographic order. This requires a total order on the letters in the alphabet, which is then extended onto words.

[1] The field of epigenetics differentiates between even more kinds, which are not relevant for our analysis.

[2] In our analysis we do not care about every l-mer, but only the non-overlapping ones; thus, n/l.

i	SA	$S^{SA[i]}$
0	4	AAGG
1	0	AAGTAAGG
2	5	AGG
3	1	AGTAAGG
4	7	G
5	6	GG
6	2	GTAAGG
7	3	TAAGG

Figure 3.1: A suffix array for the string AAGTAAGG. The suffixes are usually not stored explicitly, but shown here for didactical purposes. Also, the empty suffix—sometimes written as ε or λ—is ignored.

Definition 6 (Lexicographic Order). *Let $u, v \in \Sigma^*$ be two distinct words with $|u| \leq |v|$. Then u is called lexicographically smaller than v if*

1. *u is the empty word ($|u| = 0$),*
2. *u is a prefix of v ($u = v[0..|u|]$), or*
3. *$\exists n \geq 0 \colon u_n < v_n \wedge \forall i < n \colon u_i = v_i$.*

Given a word S, let $S^j = S[j..]$ be the jth suffix. Then the suffix array SA is defined as $SA[i] = j$ with $\forall k < i : S^k < S^j$ i. e., position i stores the index of the ith smallest suffix. Figure 3.1 displays such a suffix array for the string AAGTAAGG.

Given the suffix array for a subject sequence S, one can look up a query q via a binary search in time $O(l \log n)$. Instead of executing a full string comparison $O(l)$ at each step, one can remember the prefix of q already matched (see Listing 3.1). This does not speed up the theoretical time bound, but is useful in practice [Grossi, 2011]. To make the search independent of the size of S, additional data structures are introduced in Section 3.2.

A suffix array can be constructed in time $\Theta(n)$ with $O(1)$ auxiliary workspace. Further details are discussed in Chapter 4.

3.2 Enhanced Suffix Arrays

In the previous section SAs were introduced. So far they allow us to match a query q against a subject S in $O(l \log n)$ time with $O(n)$ preprocessing. In order to improve the matching step, the SA is enhanced with additional information. One useful data structure is the longest common prefix (LCP) array. For each entry i in the SA the LCP holds the length of the longest common prefix between the suffixes $S^{SA[i]}$ and $S^{SA[i-1]}$.

```
1  fn find_matches
2  requires S, SA
3  input q
4
5  let upper ← |S|
6  let lower ← 0
7  let upper_i ← 0
8  let lower_i ← 0
9
10 // do a binary search
11 while lower ≠ upper do
12     let mid ← (upper + lower) / 2
13     let i ← min( lower_i, upper_i)
14
15     // find the common prefix
16     while S[SA[mid]][i] = q[i] do
17         i ← i + 1
18         if i >= |q| then
19             output mid
20         end
21     end
22
23     // compare the new middle to q
24     if S[SA[mid]][i] < q[i] then
25         upper ← mid
26         upper_i ← i
27     else
28         lower ← mid
29         lower_i ← i
30     end
31 end
32
33 if S[SA[lower]..SA[lower]+|q|] = q then
34     output lower
35 else
36     output ⊥
37 end
```

Listing 3.1: This algorithm matches a query q to a SA in $O(l \log n)$ time. It is improved over a binary search, in that the algorithm remembers the prefix of q, which has already been found and avoids recomparison of its characters [Manber and Myers, 1990].

i	SA	LCP	$S^{SA[i]}$	lcp − intervals
0	4	−1	**A**AGG	0
1	0	3	AAG**T**AAGG	1
2	5	1	A**G**G	3
3	1	2	AG**T**AAGG	2
4	7	0	**G**	1
5	6	1	G**G**	
6	2	1	G**T**AAGG	
7	3	0	**T**AAGG	
8		−1		

Figure 3.2: The enhanced suffix array for the string AAGTAAGG. It includes the SA and the LCP array. Note that the LCP array has one more entry than the SA. The common prefix of a suffix w. r. t. its predecessor is underlined. The character thereafter, the first variant character, is set in bold.

Definition 7 (Longest Common Prefix)**.** *Let SA be the suffix array over a string S. Then the LCP values are defined as*

$$LCP[i] := \max\left\{m \mid \forall j \leq m : S_j^{SA[i]} = S_j^{SA[i-1]}\right\} .$$

For convenience, the first and the nth entry in the LCP array are set to -1. The compound structure of an LCP array with a SA is called enhanced suffix array (ESA). Figure 3.2 shows an example ESA.

The theoretically fastest sequential algorithms, creating an LCP array from a SA, have an asymptotic time complexity of $O(n)$ [Kasai et al., 2001, Manzini, 2004]. They require $\Theta(n)$ additional memory besides the space for the LCP, SA and the string. In practice they are much faster than the accompanying SA construction.

The LCP array also allows the definition of *lcp-intervals*, written $l - [i, j]$, meaning that all suffixes in $SA[i..j]$ share a single common prefix of length l. For example, in Figure 3.2, all suffixes in the interval $[4, 6]$ start with G.

Definition 8 (LCP-Intervals, from [Abouelhoda et al., 2002])**.** *Given $0 \leq i < j < n$, then $l - [i, j]$ is an* lcp-interval of lcp-value l *if*

1. $LCP[i] < l$,
2. $\forall k \in [i + 1, j] : LCP[k] \geq l$,
3. $\exists k \in [i + 1, j] : LCP[k] = l$, *and*
4. $LCP[j + 1] < l$.

Given an interval $[i, j]$ the length l is the smallest number in $LCP[i + 1..j]$. This yields the definition of range minimum querys (RMQs) in the following Section.

3.3 Range Minimum Queries

The lcp-intervals created the need of finding the smallest value within a subarray from the LCP values. This can defined as follows.

Definition 9. *Let A be an array of integers. Then a range minimum query (RMQ) is the smallest element from a subinterval $[i, j] \subseteq [0, |A|)$,*

$$RMQ_A(i, j) := \arg\min_{i \leq k \leq j} A[k] \,.$$

With a naive implementation, each RMQ would iterate the interval and return the smallest element. In the worst case this requires $\Theta(j - i) = \Theta(n)$ time. Fortunately, there exists a strategy with $O(n)$ preprocessing time, which allows subsequent RMQs to be answered in time $O(1)$ [Fischer and Heun, 2007]. This additionally requires look-up tables of size $\Theta(n)$. The algorithm for constant time RMQ is beyond the scope of this thesis. The interested reader is referred to [Fischer and Heun, 2007] or [Ohlebusch, 2013] for a detailed explanation. In the context of ESAs, a RMQ is always applied to the LCP array.

Using RMQs, lcp-intervals can be computed easily. Listing 3.2 displays the algorithm `get_interval` that, given an interval for a common prefix, finds subintervals for the next letter. Subsequent calls to `get_interval` with the starting interval $0 - [0, |S|]$ allow for matching a query letter by letter to the subject. Each call takes time $O(|\Sigma|)$, so in total $O(|q| \cdot |\Sigma|)$ steps need to be taken to solve the string-matching problem for a query q. The matching step is now independent of the length of the subject with $O(n)$ additional runtime and $\Theta(n)$ memory for the creation of the ESA.

3.4 Child Arrays

The *child array* is an alternative to RMQs [Abouelhoda et al., 2004]. As can be seen in Figure 3.2, the lcp-intervals are nested and do not overlap; thus conceptually, they form a tree (compare *suffix tree* in Section 3.7). To allow its fast traversal, as with RMQs, a *Super-Cartesian tree* is built.

Definition 10 (Super-Cartesian tree, taken from [Ohlebusch, 2013])**.** *Let $A[l..r]$ be an array of integers. The* Super-Cartesian tree *$\mathcal{C}(A[l..r])$ is recursively constructed as follows:*

- *If $l > r$, then $\mathcal{C}(A[l..r])$ is the empty tree.*
- *If $l \leq r$, then the minima of $A[l..r]$ appear at positions $p_1 < p_2 < \cdots < p_k$. In this case, create k nodes $v_1, \ldots, v_k$ and label each v_i with p_i. Node v_1 is the root of $\mathcal{C}(A[l..r])$. For each j with $1 < j \leq k$, the node v_j is the right sibling of node v_{j-1}. Recursively construct $\mathcal{C}_1 = \mathcal{C}(A[l..p_1 - 1]), \mathcal{C}_2 = \mathcal{C}(A[p_1 + 1..p_2 - 1]), \ldots, \mathcal{C}_{k+1} = \mathcal{C}(A[p_k + 1..r])$. For each $i \in [1, k)$, the left child of v_j is the root of $\mathcal{C}_j$. The left and right children of v_k are the roots of $\mathcal{C}_k$ and $\mathcal{C}_{k+1}$, respectively.*

Note that a node has either a right child or a right sibling, but not both. Figure 3.3 shows a Super-Cartesian tree for the LCP array from Figure 3.2. As each node has exactly one ingoing edge, the whole tree can be represented using a *child array* with n entries. This array can be created in time $O(n)$ with $o(n)$ auxiliary workspace [Ohlebusch, 2013, p. 109].

```
1  fn get_interval
2  requires S, SA, LCP, RMQ
3  input (l−[i .. j ], m), a
4
5  do
6      if S[ SA[m] + l] ≤ a then
7          i ← m // continue in the upper half
8      else
9          j ← m − 1 // continue in the lower half
10     end
11
12     if i = j then
13         break // ‘a‘ not found, exit early
14     end
15
16     m ← RMQ(i + 1, j)
17 while LCP[m] = l // loop over all subintervals
18
19 if S[SA[i] + l] = a then
20     l ← LCP[m]
21     output (l−[i .. j ], m)
22 else
23     output ⊥
24 end
```

Listing 3.2: The procedure `get_interval` takes three parameters, an lcp-interval, a special value m and a character a. It returns the subinterval with all strings whose character at position l is a.
The procedure is a binary search over all possible subintervals one level deeper. If the interval for a is found, that is returned, otherwise the null interval $\perp$ is returned.
The parameter m is the first middle for the binary search. Additionally, the last middle, one that is a level deeper, is returned. This strategy allows the reuse of RMQs and thus, speeds up the code (compare [Ohlebusch, 2013, p. 118]).
Note that in Line 20 the l value of the new lcp-interval is not necessarily $l + 1$ (one level deeper) but $LCP[m]$, which may be bigger than that.
The loop runs at most $O(|\Sigma|)$ times, depending on the RMQ implementation used. All other operations, including the RMQ are constant, thus, the total runtime is also $O(|\Sigma|)$.

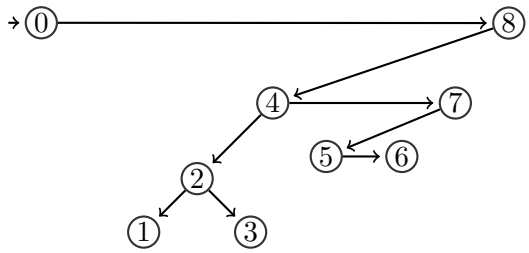


Figure 3.3: The Super-Cartesian tree for the array $-1, 3, 1, 2, 0, 1, 1, 0, -1$. Each node represents a border of an lcp-interval in Figure 3.2.
Nodes 0 and 8 form the $0-[0,7]$ interval. Its two subintervals of prefix length 1 are created by the nodes $0, 4$ and 7 corresponding to $1-[0,3]$ and $1-[4,6]$. All deeper subintervals can be iterated likewise in a top-to-bottom manner.

The child array (CLD) may be used as an alternative to RMQs in the `get_interval` function. A modified version can be found in Section A.3. Its runtime is $O(|\Sigma|)$, just as the RMQ version.

3.5 First Variant Character

Consider Line 6 from the `get_interval` method (Listing 3.2).

if $S[\ SA[m] + l\ ] \leq a$ then

That is already an optimized version of the following.

if $S[\ SA[m] + LCP[m]] \leq a$ then

Even though this code is constant in theory, it is far from optimal in practice. The reason is that S and SA are large; usually n and $4n$ byte.[3] Even for small bacterial genomes of multiple Mbp, they require megabytes of memory. Thus, the ESA does not fit into a CPUs cache and instead, parts of it are stored in main memory.

Caches are most efficient if memory is accessed in a predictable and sequential manner. In the above code, m increases until a new subinterval is found and then the search is recursively continued within. So with each call, the amount of memory accessed, is reduced. This means, the lookup $SA[m]$ is well optimized. Unfortunately, by definition, $S[SA[m]]$ is not predictable and lookups for sequential m, are almost never sequential. This renders the caching strategies used by common central processing units (CPUs) useless, resulting in cache misses and stalled instructions.

Fortunately, all values, except for m, are known in advance and thus, I propose to precompute the value of the expression $S[\ SA[m] + LCP[m]]$. It shall be called the first variant character (FVC) array, as $S[\ SA[m] + LCP[m]]$ is just one character past the LCP[4] and thus, varies between the current suffix and its predecessor. In Figure 3.2 the FVC is printed in bold face type.

[3] Assuming the test S contains only ASCII characters and SA is implemented using 32 bit integers.
[4] Remember that all indices in this thesis are zero-based; Hence the index for the character past the LCP of length l is l.

Definition 11 (First Variant Character). *The FVC is an array of length $|S| = n$ with characters from the extended alphabet $\Sigma \cup \{\bot\}$. The first entry is the special value $\bot \notin \Sigma$. For $1 \leq m < n$ the FVC is defined as*

$$FVC[m] = S^{SA[m]}[LCP[m]] .$$

The FVC array overcomes the problem of memory locality. Its entries are small—usually one byte—and accessed in a dense manner, resulting in better memory locality and finally reduced access times. Measurements for the expected performance improvement are presented in Section 6.6.

The FVC array is designed to optimize Line 6 from Listing 3.2. Unfortunately a simple replacement cannot be used for the Line 19.

```
if S[SA[i] + l] = a then
```

The reason for this is that l does not have to equal $LCP[i]$, because i is the beginning of an interval. In fact, i may be the beginning of multiple lcp-intervals, but only one FVC can be stored. In all other cases however, we can apply our optimization leading to the following code.

```
let c ← FVC[i]
if LCP[i] ≠ l then
    c ← S[SA[i] + l]
end
if c = a then
```

The definition of the FVC can be immediately converted into a construction algorithm (see Section A.2 in the Appendix). This algorithms has a runtime of $\Theta(n)$ and $O(1)$ auxiliary workspace. Other algorithms, based on [Kasai et al., 2001] and merging with the LCP computation, are conceivable. But these strategies turn out to be slower than the naive implementation (see Section 6.6).

3.6 Matches and Anchors

In the previous sections various techniques were established to solve the *substring-matching* problem. But in our comparison method for genomes, the length of a substring to match is not known in advance (see Chapter 5). So instead, the following *longest match problem* has to be solved.

Definition 12 (Longest Match Problem). *Let S, Q be strings over a common alphabet and $0 \leq i < |Q|$ be given. Find the biggest l so that $Q^i[0..l]$ is a substring of S.*

The solution to that problem is a prefix p of Q^i which is also a substring of S. Since by definition p cannot be extended by another character to the right (otherwise it would no longer be a substring of S), it is called *right maximal*. As no restrictions are applied to i, the starting point of p, the latter need not be left maximal. Additionally, p is defined to be unique, if it matches exactly once in S.[5]

[5] This differs from a common definition of a *maximum unique match* (MUM) in that the latter also requires the match to be unique in Q [Ohlebusch, 2013].

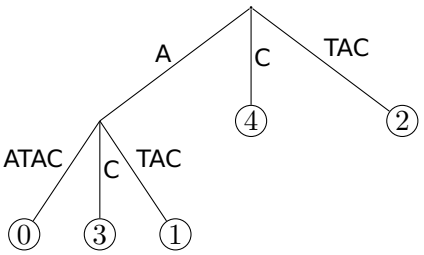


Figure 3.4: This is a suffix tree for the string AATAC. Each leaf represents a suffix and is labeled with its starting position. The leaves are sorted in lexicographic order from left to right.

Definition 13 (Anchors)**.** *Let p be a right maximal match. If it is unique in S and of some minimal length L, it is termed an* anchor *[Haubold* et al.*, 2014].*

Computing matches and anchors is straightforward with the techniques established in the last sections. Preprocess S to compute its ESA and call the `get_match` procedure of Listing 3.3, which subsequently calls `get_interval` at most once for each character. Thus, the resulting runtime is $O(|p| \cdot |\Sigma|)$, with p being the longest prefix of Q^i matching in the subject S.[6]

The return type of `get_match` is a lcp-interval. Though it may not be apparent, more than one match can be encoded by this interval or even zero if the character Q_i never appears in S. Consider the subject AAGTAAGG and the query AAGA. The result to the call `get_match`(Q^0) is $3 - [0, 1]$. Both positions $S^{SA[0]}$ and $S^{SA[1]}$ feature the common prefix of length 3 with Q but neither can be extended by another character (compare Figure 3.2). Hence the computed match is not unique.

3.7 Suffix Trees

Using an ESA and the accompanying procedures is not the only index structure to solve the *longest match problem*. The invention of the ESA is predated by the *suffix tree*. The latter is a tree of a string S where each leaf represents a suffix S^i in the sequence. The edges in the tree are labeled with substrings of S, so that each path from the root to a leaf is just the suffix S^i. Figure 3.4 shows a suffix tree for the string AATAC.

The time and space complexity for creating a suffix tree is exactly the same as that for an ESA (both $O(n)$). Furthermore, both data structures can be used to solve the string matching and the longest prefix problem, with identical complexity [Ohlebusch, 2013]. But in practice, ESAs have a lower memory requirement, which is crucial in whole-genome comparisons. So in recent applications the ESA has replaced the suffix tree as the data structure of choice [Abouelhoda et al., 2002].

[6] The implementation for `get_match`, as given in Listing 3.3, is required to be called with $Q[i..]$ as the query, to correctly solve the longest match problem. Also, to satisfy the stated time complexity, computing the length of a string has to be an $O(1)$ operation. This can be achieved by storing the length explicitly along with the characters.

```
1  fn get_match
2  requires S, SA, LCP, RMQ
3  input Q
4
5  let m ← RMQ(0, |S|)
6  let I ← (0−[0, |S|], m)
7
8  let k ← 0
9  let q ← |Q|
10
11 // Loop over the query until a mismatch is found
12 do
13     I ← get_interval(I, Q[k])
14
15     // If the match cannot be extended further, return.
16     if I = ⊥ then
17         output I
18     end
19
20     (d−[i, j], m) ← I // Destructuring assignment
21
22     l ← q
23     if i < j and d < l then
24         l ← d // Reduced RMQ
25     end
26
27     // Extend the match
28     p ← SA[i]
29     while k < l do
30         if S[p+k] ≠ Q[k] then
31             output (k−[i, j])
32         end
33         k ← k + 1
34     end
35 while k < q
36
37 output (q−[i, j])
```

Listing 3.3: This procedure computes a right maximal match. For each character `get_interval` is called at most once. So the resulting runtime is $O(n \cdot |\Sigma|)$ with a smaller constant than [Ohlebusch, 2013, p. 119] due to a lower number of RMQs.

4 Parallel Suffix Array Construction

Since the invention of suffix arrays [Manber and Myers, 1990], various suffix array construction algorithms (SACAs) have been devised [Puglisi et al., 2007]. They differ greatly in their resource consumption (see Table 4.1).

The skew algorithm was one of the first with an asymptotically linear worst-case runtime [Kärkkäinen and Sanders, 2003]. divsufsort and msufsort are among the fastest algorithms in practice [Mori, 2005, Maniscalco and Puglisi, 2006]. The worst-case runtime of radixSA is super-linear, but its expected runtime is $\Theta(n)$ [Rajasekaran and Nicolae, 2014]. The bucket pointer refinement algorithm has no known precise upper boundary for the runtime, but performs well in practice [Schürmann, 2007]. The biggest disadvantage of radixSA and BPR is their need for an additional array storing the bucket pointers, which requires additional $4n$ byte of memory.

Some of the given algorithms are not easily parallelizable (BPR), others can be parallelized using a parallel random access memory (PRAM) model, but no freely available reference implementation exists (radixSA, msufsort). Some research has been done on the parallelization of SACAs for graphics processing units (GPUs) [Kulla and Sanders, 2006, Osipov, 2012, Deo and Keely, 2013] and distributed mesh networks [Navarro et al., 1997]. But barely any effort has been made to achieve a *practical* improvement on contemporary multi-core CPUs, with the exception of divsufsort, which also features a multi-threaded version, but has poor CPU utilization (see Section 6.5). Thus, in this chapter an algorithm is introduced, which is easily parallelizable and scales well across common multi-core processors. It is a variant of a known fast and lightweight (meaning $o(n)$ auxiliary space) algorithm. In this chapter, the new algorithm is explained and analyzed in detail.

Table 4.1: Worst-Case Complexities for Various SACAs.

Implementation	Runtime	Space (byte)
skew	$\Theta(n)$	$\Theta(n)$
radixSA	$O(n \log n)$	$9n + o(n)$
divsufsort	$O(n \log n)$	$5n + O(1)$
msufsort	$O(n^2 \log n)$	$6n + o(n)$
BPR	$\Omega(n^2 / \log n)$	$9n + O(1)$

4.1 The Improved Two-Stage Algorithm

Recall from Section 3.1 that the SA of a text T contains the indices to its suffixes in lexicographic order. So the simplest algorithm for its construction is filling the SA[1] with

[1] The notation SA is used to refer to the concept of suffix arrays as introduced in Section 3.1. The in-memory representation SA, is simply an array of numbers, which may not be a valid SA in intermediate steps in a SACA; hence, the different notation.

S	G	T	G	A	G	G	T	$
Types	S^*	L	L	S^-	S^-	S^*	L	S^*

Figure 4.1: Below each character of the string GTGAGGT is the classification for the suffix starting at that position. The $ represents the end of the string and is equivalent to the NULL byte in C-style strings.

the numbers 0 to $|T|$ and then sort them according to the order of their suffixes using a suitable algorithm [Bentley and McIlroy, 1993]. That leads to $O(n^2 \log n)$ runtime in the worst case, if a multikey introsort is used [Musser, 1997]. For long texts such a runtime is unacceptable. The two-stage algorithm [Itoh and Tanaka, 1999] and its successor, the improved two-stage algorithm [Mori, 2005] reduce the number of suffixes that need to be sorted explicitly, to a subset. The other suffixes are placed into the SA by induced sorting. Variants of this algorithm have already been implemented and are in wide use [Mori, 2005, Maniscalco and Puglisi, 2006]. However, I here present the first thorough description and proof of this process in four steps. A complete pseudocode implementation is given in the Appendix, starting at Page 53.

Step 1. Initialization

First, each element in the SA is initialized to the special value $\perp$, representing an empty memory cell. Then, each suffix is classified into one of three types—Type L, S^- or S^*—according to the following definition.

Definition 14 (Suffix Types)**.** *Let T^i be a suffix and T^{i+1} its successor. Then T^i is of*

1. *Type L iff $T^i > T^{i+1}$,*
2. *Type S^- iff $T^i < T^{i+1}$ and T^{i+1} is not of Type L,*
3. *Type S^* iff $T^i < T^{i+1}$ and T^{i+1} is of Type L.*

Furthermore, the empty suffix $T^{|T|}$ is defined to be of Type S^*. Additionally, each suffix of Type S^- or Type S^* is said to be also of Type S.[2]

Figure 4.1 displays the classification of suffixes for the string GTGAGGT. This can be done in a single scan of the text (see Listing 4.1). Thereto the algorithm uses the following lemma.

Lemma 1. *Let T^i be a suffix of text T. If T^i is of*

1. *Type L then $T_i \geq T_{i+1}$,*
2. *Type S^- then $T_i \leq T_{i+1}$, and*
3. *Type S^* then $T_i < T_{i+1}$.*

[2] In the original short description [Mori, 2005] these types are called A, B and B^*. However, their definition differs from the types A and B in [Itoh and Tanaka, 1999] and instead resembles [Ko and Aluru, 2003]. Hence the latter naming scheme (L and S) is adopted here.

```
1  fn classify
2  requires T, Bucket_L, Bucket_S*, Bucket_S−
3
4  Bucket_S*[$].size ← 1
5  i ← n−1
6  goto line 18
7
8  while i ≥ 0 do
9      if T[i] ≥ T[i+1] do
10         Bucket_L[T[i ]]. size ++
11         i ← i−1
12         goto line 8
13     end
14
15     Bucket_S*[T[i], T[i +1]]. size ++
16     i ← i−1
17
18     while i ≥ 0 and T[i] ≤ T[i+1] do
19         Bucket_S−[T[i], T[i +1]]. size ++
20         i ← i−1
21     end
22 end
```

Listing 4.1: This algorithm scans the text T once from right to left. During this process the suffixes are classified and the size counter of their corresponding bucket is increased.

Proof. Case 1 and 2 follow immediately from the definition. I now prove Case 3 by contradiction.

Assume T^i is of Type S*, but $T_i = T_{i+1}$ ($T_i > T_{i+1}$ is trivially false). Then, $T_{i+1} \geq T_{i+2}$ as T^{i+1} is Type L; but equally $T_{i+2} \geq T_{i+1}$ has to hold, to satisfy the Type S* property for T^i. So now with $T_i = T_{i+1} = T_{i+2}$, the prerequisite $T^i < T^{i+1}$ transfers to $T^{i+1} < T^{i+2}$. This is a contradiction to the definition which states that T^{i+1} has to be of Type L. □

When the type of a suffix is established, it is sorted into a *bucket*. For Type L suffixes there is one bucket per character from the alphabet. For Type S$^-$ and Type S* the first two characters are used.

Lemma 2. *Let the suffix T^i be of Type L, and T^j of Type S and they begin with a common character $c \in \Sigma$. Then T^i is lexicographically smaller than T^j.*

Proof. Let c_0 be the first non c character in T^i. As T^i is of Type L it has to hold that $c_0 < c$. Similarly, for the first non c in T^j, $c_1 > c$. Let k be the smaller of the indices for these characters. Then $c_0 = T^i[k] < T^j[k] \leq c$ or $c \leq T^i[k] < T^j[k] = c_1$ holds and thus, $T^i < T^j$. If no such characters c_0 or c_1 exist, then T^i is a prefix of T^j and the inequality still holds. □

i	Bucket	SA	$S^{SA[i]}$
0	S^*[$]	7	ε
1	S^-[AG]	3	AGGT
2	L[G]	2	GAGGT
3	S^-[GG]	4	GGT
4	S^*[GT]	5	GT
5		0	GTGAGGT
6	L[T]	6	T
7		1	TGAGGT

Figure 4.2: This figure shows the SA for the string GTGAGGT. Its suffixes were sorted using the relations for their types.

Lemma 3. *Let the suffixes T^i be of Type S^* and T^j of Type S^-, beginning with the common characters c and d. Then T^i is lexicographically smaller than T^j.*

Proof. By definition, T^{i+1} is of Type S and T^{j+1} is of Type L. Using Lemma 2, their order, and the order of their predecessors can be inferred as $T^i < T^j$. □

Once the size of each bucket is computed, using the relations above, their starting points in the SA can be calculated. Finally, the indices of the Type S^* suffixes are inserted into the SA. Figure 4.2 displays the relations from Lemma 2 and Lemma 3 in the SA, for the suffixes from Figure 4.1.

Lemma 4. *The algorithm from Listing 4.1 correctly classifies all suffixes.*

Proof. By definition, every character is greater than the sentinel ($), so the algorithm places the empty suffix into its own bucket and jumps to Line 18, thus, continuing with the first regular suffix T^{n-1}.

The correct classification for all other suffixes is proven by induction over i, descending from $n-1$. Assume all suffixes, including $i+1$, have already been classified. Now the algorithm can be in one of two states:

1. T^{i+1} is of Type S and the algorithm is currently on Line 18. If T^i is of Type S^-, then the condition $T_i \leq T_{i+1}$ holds by Lemma 1, and it is sorted into its bucket. Conversely, for T^i of Type L the condition is false and the suffix is instead classified by Line 10. (The combination T^i of Type L, T^{i+1} of Type S, and $T_i = T_{i+1}$ is impossible.)

2. T^{i+1} is of Type L and the algorithm is currently on Line 8. If T^i is also of Type L, then by Lemma 1, the condition $T_i \geq T_{i+1}$ is true and T^i is also classified as Type L. However, if that condition is false, T^i has to be of Type S^*, by Definition 14.

Thus, iteratively, all suffixes are classified to a type and sorted into their buckets, until i reaches 0. □

```
1  fn induce
2  requires T, SA, Bucket_S⁻
3
4  for i=n to 0 do
5      j ← SA[i]
6
7      if j ≠ ⊥ and T[j−1] ≤ T[j] do
8          B ← Bucket_S⁻[ T[j−1], T[j]]
9          SA[B.start + B.size − 1] ← j−1
10         B.size ← B.size − 1
11     end
12 end
```

Listing 4.2: This algorithm scans the suffix array once from right to left. Each encountered suffix is checked, whether its predecessor is of Type S-. If so, the latter is placed to the end of its bucket in the SA.

Step 2. Sorting Type S* Suffixes

Now each bucket of Type S* suffixes is sorted using a string sorting algorithm. This differs from sorting integers, in that a comparison of two strings may take time $O(n)$. Even worse, the more alike two strings are, the longer their common prefix is, and thus, the longer the comparison takes.

To optimize for the *multi-key* nature of strings, a ternary-split quicksort is applied character by character [Bentley and McIlroy, 1993]. It splits the groups of strings into three sets, those whose first character is less than, equal to, or greater than the pivot. This allows the recursion in the equal part to continue with the next character and thus, avoid unnecessary recomparisons.

To avoid the worst case quadratic runtime of quicksort, it should be combined with a heapsort into an introsort [Musser, 1997]. For a bucket of m Type S* suffixes, this results in a runtime of $O(nm \log m)$.

Step 3. Induce Type S- Suffixes

The major advantage of the improved two-stage algorithm over its predecessor is the capability to induce the order of the Type S- suffixes from the Type S*. This means that fewer suffixes need to be sorted explicitly using a super-linear algorithm. Instead, Listing 4.2 shows an algorithm to place all Type S- correctly.

Lemma 5. *The algorithm in Listing 4.2 correctly places all suffixes of Type S- into the SA.*

Proof. Before the algorithm is invoked, all Type S* suffixes are already at their correct position within SA. Now let T^j be the Type S- suffix to be placed at position i. We now prove, by induction over i, that when the scan reaches $SA[p] = T^{j+1}$, the suffix T^j is correctly placed at position i, the end of its corresponding bucket.

Assume, that the scan has reached a certain i, where the Type S- suffix T^j needs to be placed, and all Type S suffixes to the right of that position are already placed correctly.

Then also the successor T^{j+1} has been encountered, at position p. For every suffix T^k from the same bucket as T^i the following holds

- $T^{k+1} < T^{i+1}$ iff $T^k < T^i$, and
- $T^{k+1} > T^{i+1}$ iff $T^k > T^i$.

Thus, when the scan reached p, all the suffixes greater than T^i have already been placed correctly. Likewise, no smaller suffix from that same bucket has yet been inserted, since their predecessors were not seen, so far. So at that moment, the bucket counter pointed at i and T^i was inserted correctly.

The initial condition for this induction is that for every Type S$^-$ suffix there is a lexicographically greater Type S* suffix, already in place. This is trivially true by Definition 14, and thus, in the SA exists a right most Type S* suffix, with no Type S$^-$ suffixes that might be lexicographically greater.

Furthermore, this algorithm does not accidentally insert a suffix of Type L. Assume a suffix T^i of Type S, whose predecessor is of Type L. Then T^{i-1} would ‚slip through' the condition in Line 7, if $T_{i-1} = T_i$ was satisfied. But since $T_i \leq T_{i+1}$, that would require T^{i-1} to also be of Type S, a contradiction. □

Step 4. Induce Type L Suffices

Finally, all suffixes of Type L are induced in a similar manner to Listing 4.2 by scanning SA from left to right: For each encountered suffix T^i, if T^{i-1} is of Type L, insert T^{i-1} into the lowest free position of its bucket.

Lemma 6. *During the scan, when the position $SA[i]$ is reached, it is already filled with the correct suffix $T^{SA[i]}$. When the whole SA is processed, all Type L suffixes are sorted in ascending order.*

Proof. We prove the lemma by induction over i. Assume, the scan has reached position i and all positions $SA[0], \ldots, SA[i]$ are already filled with the correct suffixes. This is immediately true for $i = 0$ as that position is reserved for the empty suffix.

If $SA[i+1]$ should be filled with a suffix of Type S, that was already done by Step 3. So suppose $SA[i+1]$ is a position within a Type L bucket, but not yet filled with the correct suffix. Let T^j be the suffix that should be placed there. As T^j is of Type L, T^{j-1} (which is lexicographically smaller) must have been placed in $SA[0], \ldots, SA[i]$ and hence, has already been encountered in the scan. Thus, $SA[i+1]$ will be filled once it is reached by the scan. □

Theorem 1. *The improved two-stage algorithm correctly sorts all suffixes into a SA.*

Proof. Follows from the proofs for Steps 1, 3, and 4 (Lemma 4, 5, and 6, respectively) and the correctness of the sorting algorithms, used in Step 2. □

4.2 Complexity

Memory

Apart from temporary variables, the algorithm uses the big arrays SA, T, and the buckets. Furthermore, the callstack is used for sorting in Step 2. The text T needs n memory cells

(read: byte). For every element in the SA, $\log n$ bits are needed, resulting in a total memory usage of $\Theta(n \log n)$. If $\log n$ is smaller than the word size of the CPU, the memory usage is linear (e. g., $4n$ for a $32\,\mathrm{bit}$ processor). The number of buckets is only dependent on the alphabet and thus, constant with respect to n. The sorting routines of Step 2 make heavy use of recursion and equally the callstack. In introsort the recursion depth is limited to a $\Theta(\log n)$ threshold. So, all-in-all, the required memory is $\Theta(n \log n)$ theoretically, and $5n + o(n)$ byte on standard machines.

Runtime

Let a text T of length n be given. Then the classification of all suffixes using the algorithm from Step 1 takes $\Theta(n)$ steps. Further, the calculation of the correct starting positions for all buckets is $\Theta(\sigma^2)$ and hence, $O(1)$ with respect to n.

In the worst case, every second suffix is of Type S*. Thus, $n/2$ strings need to be sorted explicitly by standard algorithms. Hence, the runtime is $O(n^2 \log n)$ for Step 2. Step 3 and Step 4 use very similar algorithms, which both iterate the SA just once. Thus, their runtime is $\Theta(n)$.

The total runtime is dominated by Step 2, resulting in $O(n^2 \log n)$ for the complete algorithm. This is the same as for the naive algorithm, but with a smaller constant.

Concurrency

As seen in the previous section, Step 2 is the part of this algorithm with the biggest influence on its runtime. Luckily, the process can be sped up by distributing the sorting of different buckets across all available CPUs. For an alphabet of size σ this results in a runtime of $O(n^2/\sigma^2 \log n)$ if the number of processors p is greater than the number of Type S* buckets $(\sigma^2 - \sigma)/2$ (see Lemma 1) and the buckets are equally filled. In the worst case, when all Type S* suffixes start with the same two characters, no improvement can be achieved. However, the probability that two random Type S* suffixes start with the same two characters is just $\frac{2}{\sigma^2 - \sigma}$ (assuming uniform distribution of characters).

To provide more concurrency, when the number of processors exceeds the number of filled buckets, the latter may be split into subbuckets, by a quicksort over the first character. Multiple runs may prove useful for large p. Using this approach, the total runtime is reduced to $O(n^2/p \log n)$ for $p \ll n$.

This method of parallelization can be implemented on a PRAM with little communication overhead. The concurrent read exclusive write (CREW) nature of this algorithm produces close to no need for expensive cache invalidation across processors. Thus, it is an excellent candidate for implementation on standard multicore machines.

4.3 Implementation

As a *proof of concept*, I implemented the parallelized improved two-stage algorithm in the psufsort package. Its C++11 sources are available as free software on GitHub.[3]

[3] https://github.com/kloetzl/psufsort

In addition to the library, psufsort comes with a wrapper program, which computes the SA of a given file. Furthermore, the result is validated with a routine from libdivsufsort, to check its integrity.

psufsort was created to replace libdivsufsort in the *low memory* mode of andi (see Section 5.5). Starting with version 0.9, the former can be activated using a compile-time switch. Results on the performance of psufsort can be found in Section 6.5.

5 The Anchor Distance

As seen in Chapter 2, evolutionary distances are a widely used basis to create phylogenies [Felsenstein, 2004]. Various alignment-free methods for computing distance have been developed over the years and some of these have been described in Section 2.2. During the development of the anchor distance we focused on high accuracy at great speed, even under strict resource limitations. In the following sections, our approach is explained in detail.

5.1 Definition

In Chapter 2, an *evolutionary distance* was defined as a function $d\colon M \times M \to [0, \infty)$, where M is a set of genomic sequences. The anchor distance is computed from two sequences, one called the *subject* S and a *query* Q with $Q, S \in \{A, C, G, T\}^*$[1]. Due to evolutionary events like *gene duplication*, the comparison with our yet-to-be-defined anchor distance d_{asym} may not be symmetric (i. e., $d_{asym}(Q, S) \neq d_{asym}(S, Q)$). To overcome this limitation, the final distance is the average of both comparisons.

Definition 15 (Anchor Distance)**.** *Let S_1 and S_2 be two genetic sequences. Then the* anchor distance *is the average of the two asymmetric comparisons with d_{asym}.*

$$d_a(S_1, S_2) = \frac{d_{asym}(S_1, S_2) + d_{asym}(S_2, S_1)}{2}$$

To compute the asymmetric anchor distance of S and Q, generate the ESA of the subject, concatenated with its *reverse complement*. Then Q is streamed against S as follows. Set q to 0 and continue until it runs past $|Q|$. Compute the longest match of $Q[q..]$ with S. Continue finding matches and each time incrementing q by the length of the match until an anchor is found. Save its characteristics and keep finding matches until a second anchor is found. Unless the anchors form an *anchor pair*, replace the saved state of the first anchor with the second and try finding another second anchor. The anchors form a pair if they are equidistant, that is, their distance on Q is the same as for their counterparts on S (see Figure 5.1).

An anchor pair frames a region of nucleotide sequence, which is assumed to be homologous. Since the two sequence parts are of equal length, a Hamming distance can be easily computed. More precisely, the number of homologous nucleotides and SNPs are counted. These numbers are cumulated for every additional anchor pair found. The final anchor distance is the Jukes-Cantor corrected Hamming distance (see Section 2.2).

[1] Even though the genomic alphabet contains only the four characters A, C, G and T, the actual alphabet used by andi has the following additional characters $\{!, ;, \#, \backslash 0\}$. The $\#$ is used to separate a genome sequence from its reverse complement. Both of which might be made up of multiple contigs separated by ! and ;, respectively.

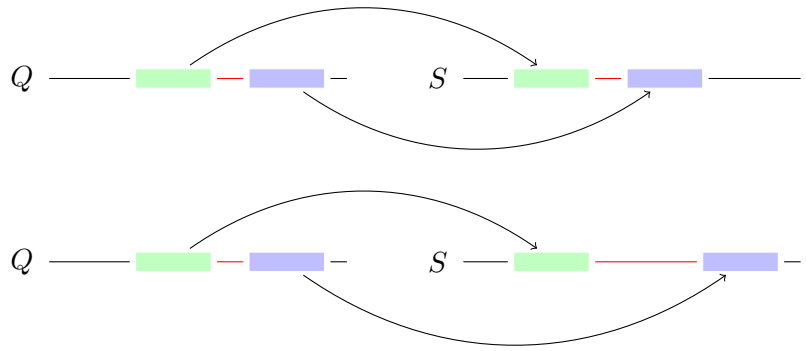


Figure 5.1: Anchor Pairs. In the upper panel the anchors on Q and S are equally spaced and hence considered a valid anchor pair. Thus, the SNPs in the framed segment, shown in red, are counted. For the second figure the anchors are not equidistant and therefore ignored.

Definition 16 (Corrected Asymmetric Anchor Distance)**.**

$$d_{asym}(S,Q) = JC\left(\frac{\#SNP}{\#HomologousNucl}\right)$$

Listing 5.1 shows the pseudocode algorithm to compute d_{asym} using the previously established procedures `get_interval` and `get_match`. Various exceptions may arise during this calculation, which need to be handled, individually.

- The query and the sequence might be identical or the former might be contained in the latter. This leads to a single match extending over the full query; $d_{asym}(S,Q) = 0$.
- With very closely related sequences only a single anchor pair might extend over the query completely; $d_{asym}(S,Q) = JC(d_H(S,Q))$.
- If the query contains a subsequence multiple times that is only found once in the subject (e. g., gene duplication), the same part of the subject might be accounted for a homologous sequence more than once. Eventually, the count for homologous nucleotides might exceed the length of the subject. In this case, the distance is set to the special error value `NaN`.
- With very diverse, or totally unrelated sequences, no anchor pair may be found. In these cases, the distance is also set to `NaN` (see Section 5.4).
- If an anchor could serve as both, a left and right anchor, be sure to count its nucleotides only once, to avoid biasing the result.

In addition to the anchor distance, `andi` computes another characteristic, the *coverage*, that is, the relative amount of homologous nucleotides. This is useful for debugging, but not accurate enough to serve as a distance in its own right.

```
1  fn dist_anchor
2  requires S
3  input Q
4
5  let E ← ESA(S)
6  let L ← threshold(S,Q)
7  let Snps ← 0
8  let Homol ← 0
9
10 let last_pos_q ← 0
11 let last_match ← ⊥
12 let last_was_right_anchor ← false
13
14 let q ← 0
15 while q < |Q| do // Stream the complete query
16
17     // Find the next match
18     m ← get_match(E, Q[q...])
19     if m.isUnique and m.length ≥ L then
20
21         // m is an anchor
22         if q − last_pos_q = m.pos − last_match.pos then
23             // We have found a pair
24             Snps ← Snps + count_diff(Q[last_pos_q...q], S[last_match.pos...m.pos])
25             Homol ← Homol + q − last_pos_q
26             last_match ← m
27             last_was_right_anchor ← true
28         else
29             // Correctly count the nucleotides from right anchors
30             if last_was_right_anchor = true then
31                 Homol ← Homol + last_match.length
32             end
33
34             last_was_right_anchor ← false
35         end
36
37         // Cache values for later
38         last_pos_q ← q
39         last_match ← m
40     end
41
42     // Skip the mutation
43     q ← q + m.length + 1
44 end
45
46 output Snps/Homol
```

Listing 5.1: This algorithm computes the uncorrected asymmetric anchor distance of Q with respect to the subject S.

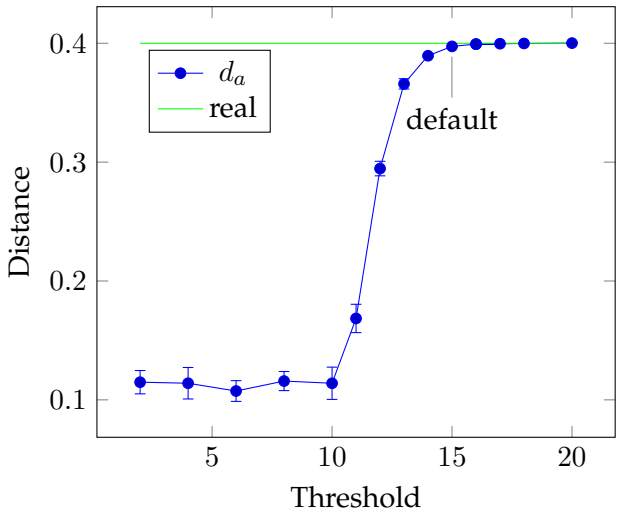


Figure 5.2: For each data point, ten sequence pairs with length $1\,\mathrm{Mbp}$ were simulated with a distance of 0.4. On the y-axis is the distance estimated by andi for the specific threshold. The default p-value of 0.05 equals a threshold of 15.

5.2 Threshold

Recall from Section 3.6 that an *anchor* is a unique match of minimum length L. Since we are interested in anchors framing homologous regions, L should be picked so that random matches are unlikely. For this, another parameter p is needed, which represents the significance of an anchor pair.

$$p = 1 - P[\text{random pair}] \tag{5.1}$$

$$P[\text{random anchor}] = \sqrt{(1-p)} \tag{5.2}$$

The probability that an anchor was found by chance alone, in Equation 5.2, depends on the length of the match. It is less likely for a long match to equal an arbitrary section in the subject than for a short match. The exact distribution of match lengths was described by [Haubold et al., 2009]. For andi, a default p of 0.05 is picked. This results in a threshold L between 10 and 16, depending on the characteristics of the compared sequences. This is much lower than the average anchor length of 60, depending on the chosen data set (here ECO29, see Section 6.4). Figure 5.2 displays the relationship between the threshold and the resulting distance.

5.3 Complexity

The requirement for fast computation of the anchor distance is low algorithmic complexity and low memory usage. Recall from Chapter 3 that computing a match to a reference ESA, takes time $O(m\sigma)$ where m is the length of the match and σ is the size of the alphabet. For our use case, Σ is the genomic alphabet, and hence, constant. Every nucleotide of

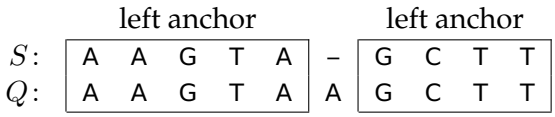


Figure 5.3: This figure shows a worst case for the *anchor strategy*, where anchors are found, but are not equidistant and thus, do not form a proper pair. The gap »–« does not exist in the data but is shown here for improved clarity.

Figure 5.4: The anchor pair frames a sequence of four nucleotides. As can be seen in the alignment, it contains two gaps. However, the Hamming distance does not see the gaps and instead counts three substitutions.

the query is matched against the subject exactly once, leading to a runtime of $O(n)$.[2] In the worst case, every nucleotide is touched again for the computation of SNPs. This still requires time $O(n)$ and $O(1)$ auxiliary working space.

The most time-consuming step is the creation of the ESA for the subject. As shown in Chapter 3, computing an SA, LCP, FVC and RMQ can be done in linear time, of which the SACA takes longest, in practice. The memory requirement is $\Theta(n)$ for the ESA.

If more than just two sequences need to be compared, multiple queries can be streamed against the same ESA. If k sequences are compared, streaming all queries against one subject takes $O(n)$ time for the ESA construction (in theory) and $O(nk)$ time for comparison. With each sequence being a subject, computing the complete distance matrix is $O(nk^2)$ with $\Theta(n)$ working memory for the anchor distance, $O(nk)$ for the sequence data, and $O(k^2)$ for the matrix.

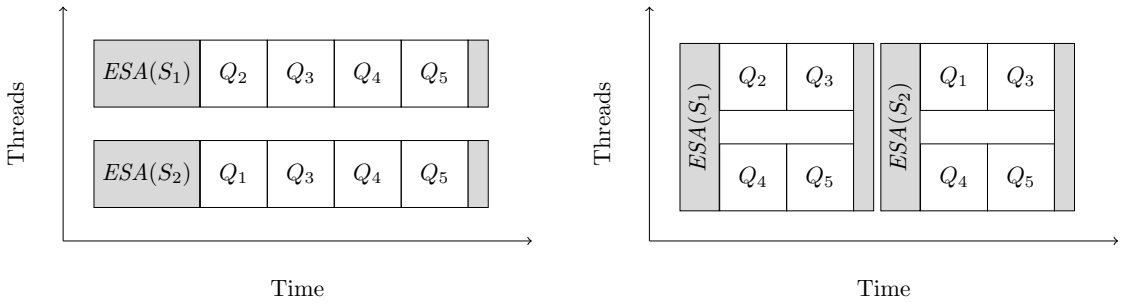
5.4 Worst Case Estimations

The *count_diff* function in Listing 5.1 computes a *Hamming distance*. This means it cannot detect indels. To protect against this, anchor pairs are required to be equidistant. This strategy leads to the following two problems.

In the example shown in Figure 5.3, the query contains one more character than the subject. When $d_{asym}(S, Q)$ is called, the first found anchor is AAGTA. Then the assumed substitution A is skipped and the second anchor is GCTT. Both anchors are unique and pass the threshold, which shall be 2, for the sake of this example. Unfortunately though, the anchors are not equidistant on both sequences and thus, no Hamming distance for the framed nucleotide(s) can be computed.

The alignment from Figure 5.4 has twice the previously described problem. The middle part, framed by the two anchors, is optimally aligned using two gaps. Unfortunately, the Hamming distance counts three substitutions instead. This way, indels, which cancel themselves out, could lead to great inaccuracies. The effects of this are evaluated in Section 6.2.

[2] W.l.o.g. $|S| = |Q| = n$ is assumed.



(a) Parallelization along subjects

(b) Parallelization along queries

Figure 5.5: The two subfigures show the two different possible modes of parallelization.

5.5 Concurrency

Assume a comparison of five genome sequences. Then the calculation of $d_{asym}(1, 2)$ is independent of $d_{asym}(3, 2)$. So theoretically, they can be run in parallel; in fact, all comparisons for different subjects may be run in parallel. Comparisons against the same subject, however, have to await the precomputation of ESA and end in its destruction.

At the current state of technology, multi-core processors are widely available. The number of processors p per machine ranges from two for smartphones, eight for standard home computers up to 64 for servers. Even though, p has been growing rapidly in the past years, it is usually still smaller than k.

If $p \leq k$ then the computation of the distance matrix is *embarrassingly parallel*. Two modes become apparent: computing multiple rows in parallel with each entry sequential or sequentially computing the entries of the rows in parallel (Figure 5.5). The first mode computes multiple ESAs in parallel and then streams all queries against them. Thus, it can be thought of parallelization along different subjects. This reduces the runtime to $O(nk^2/p)$, for $p < k$, at an increased memory usage of $O(nk + np + k^2)$.

The second mode is algorithmically more challenging, as it requires the ESA to be built in parallel. But its advantage is that it holds only the ESA for a single sequence in memory, instead of p, and thus, it uses less memory—$O(nk + n + k^2)$—which is identical to the sequential case. The runtime is likewise reduced to $O(nk^2/p)$ in theory, where parallelization of the SACA has the biggest impact in practice.

5.6 Implementation

The anchor distance d_a can be implemented using the generic match finding tool vmatch [Kurtz, 2014]. However tests have shown that our own implementation, andi, is up to seven times faster, even for small data sets. In this section we explain, how andi achieves its speed.

Software Engineering

The reference implementation for the anchor distance, andi, is written in C/C++. Its sources are released on GitHub[3] as *free software* under the GNU GENERAL PUBLIC LICENSE VERSION 3 [Free Software Foundation, 2007]. Prebundled packages using autoconf are also available, with the latest release being v0.8.1 at the time of writing.

To provide good code readability, every function is documented with doxygen style comments. The correctness of the code is constantly monitored with unit tests by the *continuous integration framework* Travis CI. The unit tests achieve a coverage of more than 80% for all relevant lines.[4] Most of the uncovered lines are handling exceptions (e. g., failed allocations). To prove correctness even under exceptional circumstances, the code was statically analyzed by the scan-build utility from the LLVM framework [Lattner and Adve, 2004].

I/O Formats

andi is designed—following the Unix philosophy—to work with plain text data formats. As input, the Fasta format was chosen for its simplicity and wide application in biology.

```
>S1
AAGTAAGG
>S2
AACTACGG
```

Each line starting with a >, marks the header line for a new sequence, which contains its name. All subsequent lines are its DNA. If a file contains more than one sequence, it is called a *multi*-Fasta file.

The output of andi is a distance matrix, for which the Phylip representation was chosen, used by a lot of bioinformatics software [Felsenstein, 2005]. On the first line, the size of the matrix is given. Then follows a line for each sequence, starting with its name, followed by the distances.

```
2
S1␣␣0␣␣0.2
S2␣␣0.2␣0
```

Concurrency

In Section 5.5 the two possible modes of parallelization were explained; andi implements both. By default, it computes rows in parallel using as many threads as requested by the `-t` command line switch. Using the `--low-memory` flag, andi can be switched into the other mode, where it only holds the ESA for one sequence in memory, hence the name. Both modes are implemented with the OpenMP framework.

[3] The official Git repository for andi can be found under `https://github.com/EvolBioInf/andi`.

[4] Blank lines, comments, and statements spanning multiple lines are considered irrelevant. For details visit `https://coveralls.io/r/EvolBioInf/andi`.

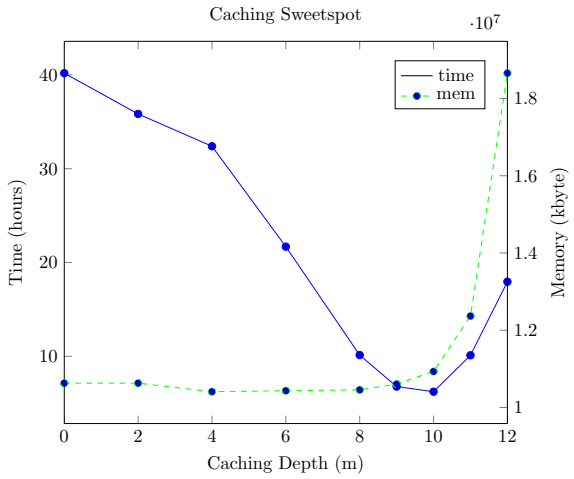


Figure 5.6: Caching Characteristics. Only a single run was made for each data point, hence the slight fluctuation in memory usage (dashed curve).

Caching

Let S be a subject sequence from a large data set. Then for each new match w. r. t. this subject, the `get_match` search starts with a global RMQ (see Line 5 from Listing 3.3). Even though a RMQ is a $O(1)$ operation, it still takes up to 45 CPU cycles.[5] So for all matches starting with an A, multiple RMQs and memory lookups are executed, with always the same result.

To avoid recomputation of intervals for identical match prefixes, a cache is introduced. This cache is a simple table which maps a prefix Σ^m of length m to an lcp-interval from which the `get_match` procedure may continue its search.[6] The table itself can be filled efficiently using a recursive version of `get_match`.

For andi a prefix length of $m = 10$ has been proven to achieve the best speedup across a wide variety of data sets, from 20% for small sets, up to 6.8-fold for a set of 3085 *S. pneumoniae* genomes (see Section 6.4).

Figure 5.6 shows the runtimes and memory consumption of andi for different caching depth on the same data set (see Section 6.4). It can be seen that for $m = 10$ andi is fastest, with little additional memory. Thus, that value has been picked as default.

[5] Measured with valgrind [Nethercote and Seward, 2007].
[6] This is similar to the bcktab table by [Abouelhoda et al., 2004].

6 Results

Bioinformatics software is commonly evaluated by two characteristics: accuracy and performance [Filion, 2015]. The basic accuracy of andi has already been discussed in Section 2.3. In this chapter we will study the accuracy in presence of other effects, such as indels and recombination, as well as on real data. Later, we explore the performance of the distance estimators. To enable reproducibility, the computers used for comparison are defined in the next section.

6.1 Machines

A standard desktop computer running Ubuntu 14.04 LTS was used for most of the runtime measurements. It is henceforth referenced as M1. Its $64\,\mathrm{bit}$ CPU is an Intel Core i7 870 with $2.93\,\mathrm{GHz}$, capable of running eight simultaneous threads of executions. Furthermore, M1 has 7.8 GiB of random access memory (RAM) and $976\,\mathrm{GB}$ of disk space.

For bigger data sets M2 is used. It features an AMD Opteron 8356 with 32 cores, each clocked at $2.3\,\mathrm{GHz}$. It has $256\,\mathrm{GB}$ of RAM with plenty of free disk space and is running a CentOS. Computers of these sizes have become standard equipment in most labs in the past years. Good bioinformatics software should make full use of their computational capabilities.

6.2 Insertions and Deletions

We have already discussed in Section 5.4, that andi may be sensitive to indels. To further explore for this issue, we simulated pairs of sequences with a fixed distance and varying indel rate. Figure 6.1 displays the results of this test.

For each data point in Figure 6.1, two sequences of length $100\,000\,\mathrm{bp}$ were simulated with a substitution rate $\pi = 0.1$ and varying indel rate ϕ. Now there are two equally correct measures of the *evolutionary distance*; the substitution rate π and the total error rate $\pi + \phi$.

The substitution rate has been used for a long time to estimate evolutionary distances [Zuckerkandl and Pauling, 1962]. However, it remains unknown how to extend these results to indels, which may be under higher selection pressure. Also, indels are commonly clustered, because a single evolutionary event likely causes an indel spanning multiple nucleotides.

Figure 6.1 shows the ideal results for both approaches. The lower, dashed line is the constant substitution rate, whereas the upper line is the error rate, counting each substitution and each indel as a single evolutionary event. As long as the indel rate ϕ is one order of magnitude smaller than the substitution rate of $\pi = 0.1$, all methods estimate π quite well.

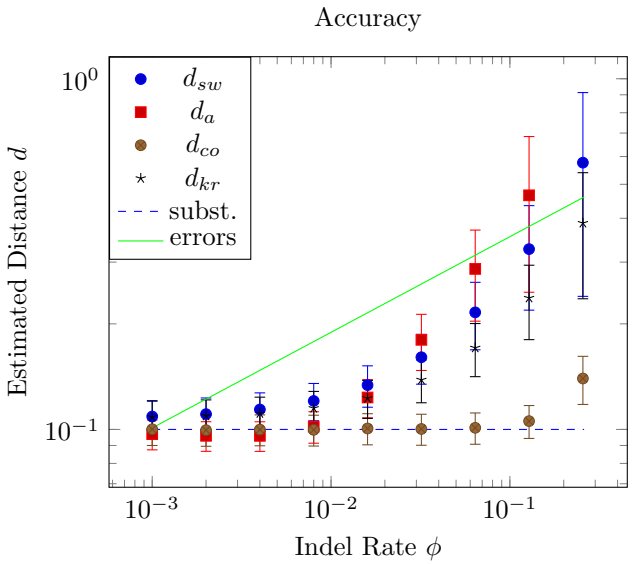


Figure 6.1: For each data point, one hundred sequence pairs with a distance of 0.1 and a certain indel rate were simulated. The mean and variance are plotted. Both theoretical distances substitutions and errors are shown as lines (continuous and dashed, respectively).

Beyond that point, all methods become increasingly upwards biased, with andi growing fastest. Its estimations rise beyond the error rate, start varying heavily and fail because of missing anchors past an indel rate of $\phi \geq 0.256$. spaced and kr show similar dynamics, but with smaller estimated rates than andi. cophylog is surprisingly resistant to indels up to a rate of 0.128. Only for $\phi = 0.256$ and thus, a total error rate of $\pi + \phi = 0.356$, do its estimations become upwards biased.

6.3 Recombination

When discussing the accuracy in Section 2.3 we assumed that substitutions are generated by a single Poisson process. In other words, the substitution rate π does not vary along or among the sequences. However, this is often not the case in real data because of *recombination* (i. e., crossover).

Recombination leads to variation in the substitution rate along a sequence. Figure 6.2 shows the local substitution rate within windows of 100 nucleotides along a recombined sequence of $1\,\mathrm{kbp}$. A good distance estimator should be resistant to recombination.

As a test, two sequences with length $1\,\mathrm{Mbp}$ were simulated with a substitution rate of 0.1 using the tool ms [Hudson, 2002]. Additionally a *population recombination rate* ρ ranging from 0.001 to 0.256 was introduced. Figure 6.3 shows the distances estimated by various methods for different levels of recombination. The distances computed by cophylog, kr, and spaced become downwards biased for increasing rates of recombination. andi is least affected by recombination. It rarely deviates more than 8% from the real distance and even gets better for higher rates of recombination. I suspect the reason for this is, it gets easier for highly clustered substitutions to find anchors in the flanking sequences.

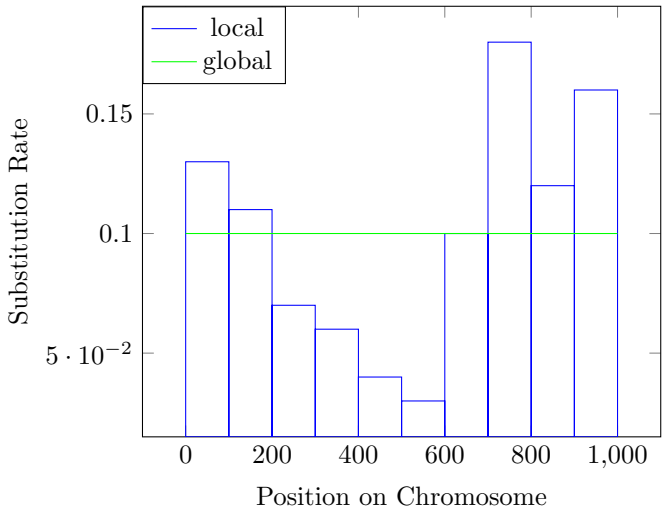


Figure 6.2: A chromosome of length 1000 was simulated with a global substitution rate of 0.1. An equal rate of recombination was introduced. This leads to fluctuations in the local substitution rate. The blue bars represent the local diversity within windows of 100 nucleotides.

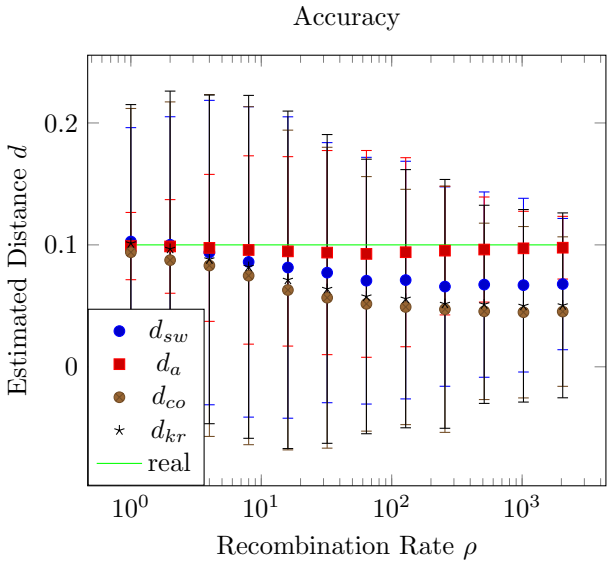


Figure 6.3: For each data point, one hundred sequence pairs with a distance of 0.1 and a certain recombination rate were simulated. The mean and standard deviation are plotted. Additionally, the straight line represents the simulated distance.

Table 6.1: Tree Metrics; All reported distances are with respect to mugsy.

	andi	kr	cophylog	spaced
rSPR	1	6	3	6
branch score	0.001739	0.013654	0.009008	0.01415

6.4 Real Data

The ultimate test for alignment-free distance estimation is its application to real data. Unlike simulated sequences, real data is riddled with surprises like indels, recombination, and sequencing artifacts. In this section we explore the usability of the various distance methods when applied to three genomic data sets.

Escherichia Coli and *Shigella*

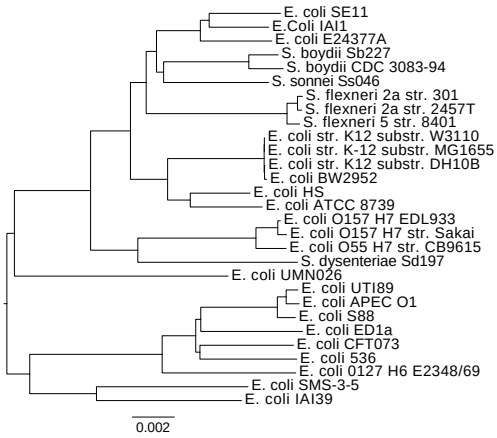
The ECO29 data set consists of 29 *Escherichia Coli / Shigella* genomes, which have previously been used for benchmarking distance methods [Yi and Jin, 2013, Haubold et al., 2014]. On average, the genomes have a length of 4.9 Mbp. As a first surprise, they contain not only the standard nucleotides A, C, G, and T, but also R, D, N and even other characters in small numbers. These stand for groups of nucleotides: R is a purine (A or G), N is any nucleotide and D means any nucleotide but C. For some implementations these need to be filtered away. The complete FASTA file for the 29 genomes comprises of 138 MB.

Figure 6.4 shows the resulting phylogenies of four alignment-free distance measures as well as an alignment-based tree as reference. All trees were computed from the distance matrices using phylip neighbor, retree and drawn with figtree [Rambaut, 2015].

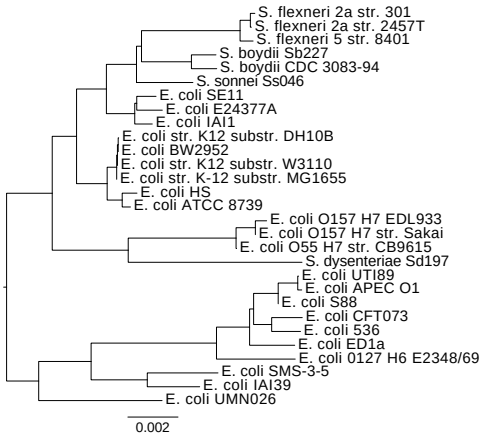
The visually worst result is computed by kr. A lot of its branch lengths differ noticeably from the reference tree by mugsy. The three phylogenies by andi, cophylog, and mugsy are quite similar and nearly indistinguishable. spaced fails to cluster four *E. Coli K12* strains together tightly.

These differences across the trees are now quantified using different metrics. First, rspr is used to compute the topological difference between each alignment-free method and the reference tree [Whidden et al., 2013]. andi has the smallest difference (1), followed by cophylog (3). As expected, kr and spaced have the worst scores (see Table 6.1). The *branch score* distance also takes the length of branches into account [Kuhner and Felsenstein, 1994]. Thus, a smaller branch score distance means, the length of two trees are more similar. Again, the tree by andi most closely resembles the reference, followed by cophylog, kr, and spaced, in this order.

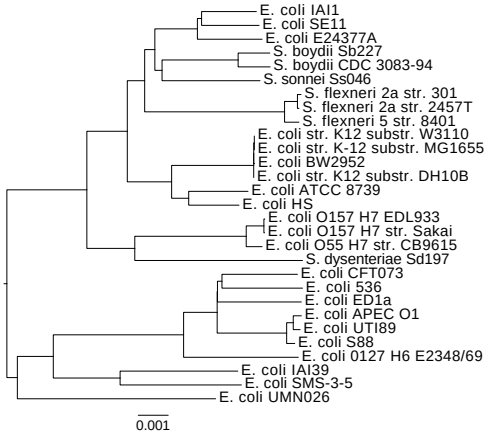
The computation of the reference tree with mugsy took 2 h, 49 min using 3 GB of memory on machine M1. The only method needing equally much memory is kr (see Figure 6.5). But kr is one and a half orders of magnitude faster with an average runtime of 5 min, 23 s. Thus, it is even faster than the multithreaded spaced, using eight cores. cophylog is slightly faster than kr, but uses only 157 MB of RAM, making it the most memory-efficient tool. The fastest tool is andi, with just 100 s for the sequential and 27.7 s for the parallel case (eight threads).



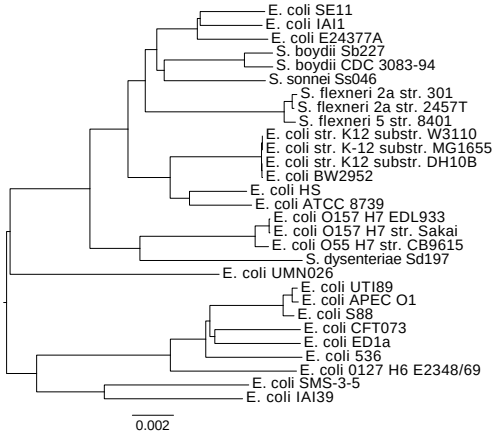
(a) andi



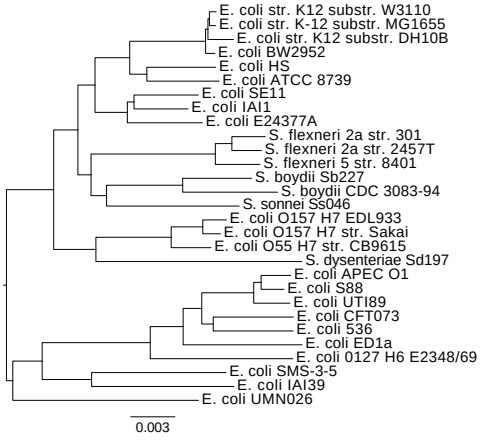
(b) kr



(c) cophylog



(d) mugsy



(e) spaced

Figure 6.4: Phylogenies for the ECO29 data set.

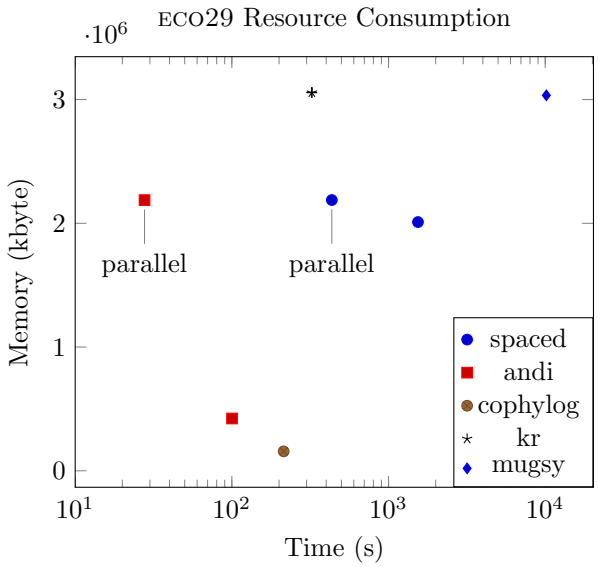


Figure 6.5: Resource consumption for the ECO29 test case. For all methods, except mugsy, the means and variance of ten runs are shown.

Roseobacter

The genus of *Roseobacter* contains highly divergent bacteria, which makes them harder to compare than *E. coli*. A set of 32 *Roseobacter* genomes was recently used to evaluate the results of alignment-free distance estimators for diverse genomes [Morgenstern et al., 2015]. A tree based on alignments of genes was used as a reference [Newton et al., 2010].

It was shown that with appropriate parameters (e. g., $k = 20$), spaced could compute a tree with an RF-distance of 25 [Robinson and Foulds, 1981], making it the most accurate method evaluated. kr scored 46 and cophylog 28 with $k = 28$. However, with default parameters, cophylog only achieves 39, which is worse than the 33, we measured for andi.

The average evolutionary distance reported by andi for the *Roseobacter* genomes is one order of magnitude higher than for ECO29 (0.22 to 0.019). At the same time, the coverage (i. e., the amount of mapped homologous nucleotides) dropped from 0.765 to 0.046. This means, the result by andi is based on only 5% of the genome. It is interesting that 5% of a genome suffice to gain an answer as good—or bad—as with competing methods.

Streptococcus pneumoniae

The largest data set used in this thesis, PNEU3085, contains 3085 genomes of *Streptococcus pneumoniae* [Chewapreecha et al., 2014]. Each of these genomes is given as several contigs, amounting to 2.2 Mbp per genome and thus, 6.8 GB for the complete data set. As all of these genomes are compared pairwise, this results in more than 4 million comparisons (9 million, if asymmetric).

It is impossible to compute distances for this data set using kr and spaced; both quickly exceed the available memory (256 GB) on machine M2. Thus, only the results for andi and cophylog can be given here. Unfortunately, no reference tree exists or can be computed via an alignment; As this data set is roughly 100 times bigger than ECO29, it needs 100^2 times

more pairwise genome comparisons. Thus, the runtime for mugsy (2 h 49 min) would explode to approximately 3.2 years, which is impractical.

The figures on page 44 show the phylogenies based on the distances computed by andi and cophylog. The most noticeable difference is the varying scale. The average distance computed by andi is 0.011 and 0.0057 for cophylog. The RF-distance between the two trees is 4544. This may seem big, but is smaller than the average distance for two random trees of that size (6166) [Haubold et al., 2014].

It took andi 6 h, 21 min and 10 GB of RAM to compute the distance matrix on M2 with 32 threads. cophylog is only single-threaded and ran for 36.5 days at just 2.3 GB. Even if cophylog supported multi-threading, andi still is approximately four times faster.

6.5 psufsort

Recall from Section 5.5 that andi has two modes of concurrency. The first computes multiple ESAs in parallel and is just a simple parallelization of the sequential case. The second mode distributes the computation of a single ESA across all available threads. Luckily, libdivsufsort, which we found to be the fastest sequential SACA, also features a multithreaded mode using OpenMP.

Figure 6.7 shows the CPU utilization for libdivsufsort with different number of threads. However, in the multithreaded case, the utilization does not rise above 128%. This motivated the search for a better parallel SACA. Out of the other algorithms listed in Table 4.1, only skew has a publicly available parallel implementation [Shun et al., 2012].

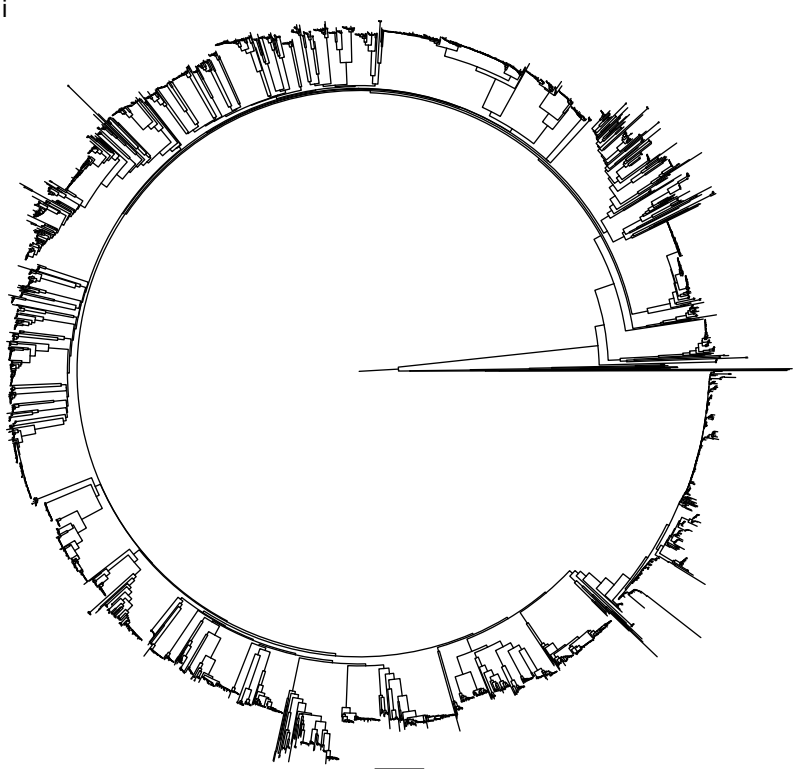
The *gauntlet* corpus [Maniscalco, 2015] was created for evaluating the performance of SACAs. It includes various files with sizes ranging from 100 kB to 15 MB. These files contain short patterns repeated very often, thus mimicking a worst case scenario for SACAs. As psufsort, unlike divsufsort, does not feature a tandem repeat detection, it cannot process the test files in any reasonable amount of time.

Another set of test files aimed at the evaluation of SACAs is the *lightweight* corpus by [Manzini and Ferragina, 2004]. Its files amount to a total size of 1 GB. Figure 6.8 shows the resource consumption of four SACAs. Of these, divsufsort and radixSA are sequential algorithms. The implementation used for skew always uses as many threads as available processors (eight on machine M1). For psufsort both the sequential and the parallel cases are shown.

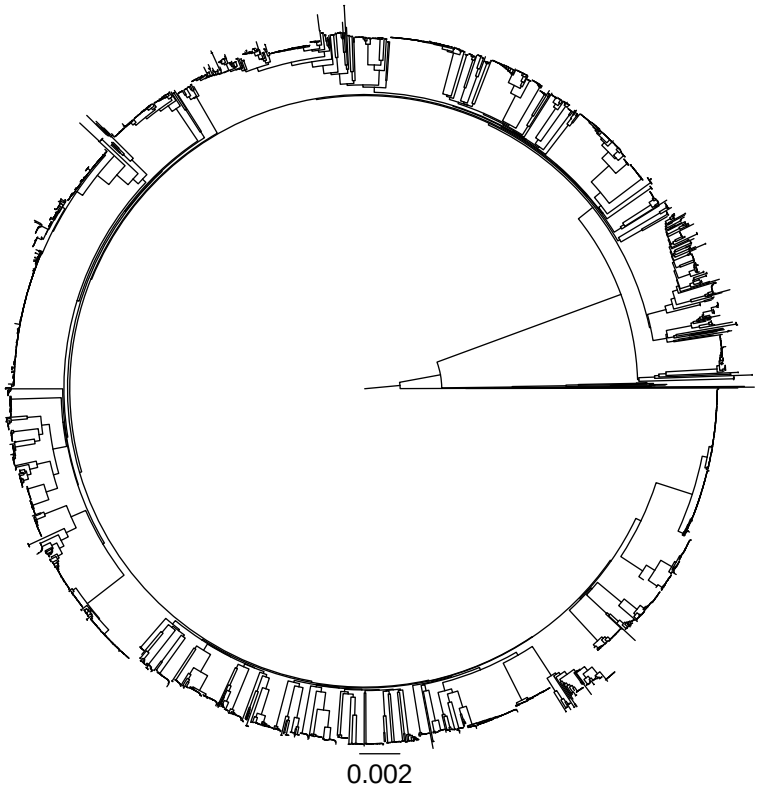
Even with eight threads, psufsort is significantly slower than the other algorithms. Interestingly, the use of eight threads do not make psufsort eight times faster, because it achieves only an average CPU utilization of 508%. divsufsort and psufsort are the only algorithms that can be considered *lightweight* as they use $o(n)$ auxiliary workspace.

To test the *practical* use of psufsort, a version of andi was created, using the former as a replacement for divsufsort in the *low-memory* mode. Compared to the normal runtime of 27.7 s, the low-memory mode takes significantly longer (59.9 s). If psufsort is used, that runtime decreases to 56.4 s, at nearly identical memory consumption. The intention of psufsort was to better utilize the CPUs, which is indeed the case, as the utilization rises from 201% to 255%.

(a) andi



(b) cophylog



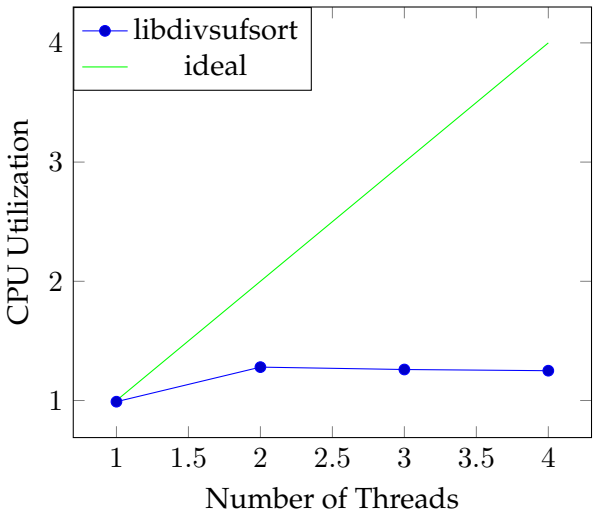


Figure 6.7: CPU Utilization of Parallel libdivsufsort

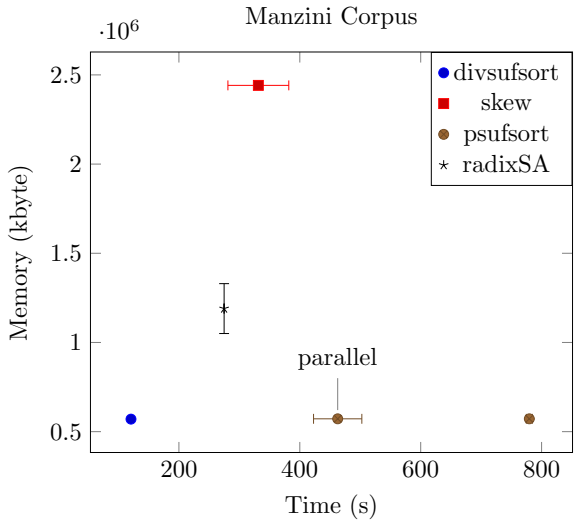


Figure 6.8: Resource Consumption for the Lightweight Corpus

Table 6.2: Performance Evaluation of Different ESA Implementations for andi

	V1		V2		V3	
	Time	Memory	rel. Time	rel. Mem.	rel. Time	rel. Mem.
simulated	1.01 s	306 MB	−4%	+3%	−27%	−29%
ECO29	28.4 s	2.1 GB	+2%	+3%	−18%	−35%
PNEU3085	7 h 34 m	10 GB	−16%	+1%	−36%	−15%

6.6 FVC Array and Child Arrays

In Chapter 3 the ESA along with multiple data structures and algorithms were introduced. However, the ESA is modular and can be build from different combinations of data structures. To achieve maximum performance for andi, multiple approaches were evaluated and the results are given in this section.

The two most important operations on the ESA are `get_interval` and `get_match` (see Listings 3.2 and 3.3, respectively). To reach the theoretical minimal bounds for these, the ESA has to be composed of the SA, the LCP array and either RMQs or the child array (CLD). The additional FVC array can be used to improve the performance in practice. Thus, for andi the performance of the following three variants were tested.

V1) SA + LCP + RMQs

V2) SA + LCP + RMQs + FVC

V3) SA + LCP + CLD + FVC

For small datasets, the runtime of andi is dominated by the construction of the ESA. Conversely, for big data sets the efficient computation of matches is important. Thus, the performance of the three variants was tested on three data sets: five simulated 1 Mbp sequences, ECO29 and PNEU3085 (see Section 6.4).

Table 6.2 shows the results for the three variants on the data sets. For V2 and V3 the difference to V1 is listed. The measurements for the simulated test case and ECO29 were run on machine M1. The figures represent the mean of ten runs. For PNEU3085 only a single run on M2 could be measured.

The resource measurements show that V3 is the fastest method and uses the least amount of memory (the latter might induce the former). It computes the distance matrix for PNEU3085 in only 4 h 49 min, using just 9.2 GB. As the complete data set of 6.8 GB is held in RAM, this means that only 2.4 GB of workspace are needed.

It can be also concluded from Table 6.2 that the novel FVC array speeds up the matching significantly (up to −16% from V1 to V2).[1] It thus can be considered useful in practice as its additional memory requirement is marginal.

FVC Construction

The FVC array can be trivially created by implementing its definition (see Section A.2). It may also be computed via a variant of [Kasai et al., 2001] and even merged with the LCP

[1] The runtime improvement from SA + LCP + CLD (not shown in Table 6.2) to V3 is −14%.

construction (listings omitted). To compare the performance of these algorithms, a small wrapper program was created, which reads files and constructs the ESA for them.[2]

As a simple test, each method had to construct the ESA for ECO29 ten times. Without an FVC (i. e., variant V1) the construction took 17.1 s on machine M1. The trivial algorithm was just slightly slower with 18.0 s, immediately followed by phi with 18.7 s. Far off was the Kasai-based algorithm with 22.9 s. All FVC construction algorithms need 2% more memory than V1.

[2]The wrapper and the algorithms are freely available at `https://github.com/kloetzl/FVC`.

7 Discussion

With the rise of high-throughput-sequencers, the number of sequenced genomes has increased rapidly over the past years. Traditional tools for genome comparison which are based on alignments are often too inefficient to handle the data available. In response, alignment-free methods have been developed over the past years. In this thesis our approach, andi, was studied in detail. In this chapter we evaluate its usefulness and make suggestions for its improvement.

7.1 Evolutionary Distances

Commonly, an alignment is used to compute evolutionary distances for genomes. andi approximates local ungapped alignments to estimate these distances. We have already seen in Section 2.3, that andi is accurate up to a simulated distance of 0.5 substitutions per site. For higher rates, no output is produced by andi. From a user's perspective, no output is better than unreliable output, as computed by kr and cophylog.

When applied to real data, andi produces satisfying results (see Section 6.4). It computes the most accurate estimations for closely related bacteria and is about as accurate as other approaches for more divergent data sets. Thus, we are confident that the tree produced by andi for the 3085 *S. pneumoniae* genomes is also highly accurate, even though, we cannot compare it to a reference tree.

We also tested some specific effects found in real data that make accurate estimations difficult, namely recombination and indels. andi is robust to recombination and provides good estimations even for high rates of recombination (see Section 6.3). In this respect, it outperforms all other alignment-free estimators.

The topologies of gapped alignments suggest that indels might lead to inaccurate estimates (Section 5.4). This was confirmed with simulations (Section 6.2). andi is accurate as long as the indel rate is one order of magnitude smaller than the simulated distance. Other tools like cophylog perform much better when applied to data with indels. Thus, it is of future interest to improve the handling of indels.

In Section 5.4 two worst-case situations were described: In the first, a single indel made anchors non-equidistant, and in the second two indels lead to overestimations of the substitution rate. The first case of non-equidistant anchor pairs can be integrated into the estimation by andi if a k-gap approach is used instead of the standard Hamming distance: Consider a non-equidistant anchor pair that is slightly off by k nucleotides, one could try to align the framed section using at most k gaps. This may be feasible is the framed section is short and k is small. We found in the ECO29 data set that if two homologous anchors are non-equidistant,[1] their distance is off by $k = 1.2$ on average. Thus, for an alignment strategy with at most two gaps, the accuracy and coverage could be increased at no significant performance overhead.

[1] Here homologous means that two anchors are reasonably distant if not equidistant.

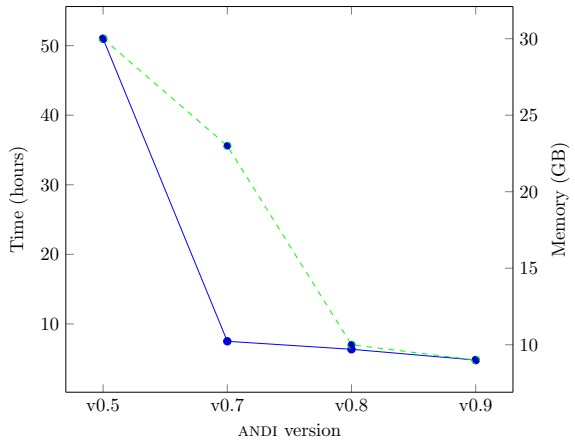


Figure 7.1: The performance of andi over time. Shown is the runtime and memory usage (dashed line) on the PNEU3085 data set.

As an alignment is significantly more complex than a Hamming distance, the previous approach has to be avoided for equidistant anchor pairs, which are much more common in the analysis. However, as already seen, indels may lead to tremendous overestimations of distances if they are located on opposite sequences. In the most basic accuracy measurements of Section 2.3 it could be observed that andi can only compute distances up to a substitution rate of 0.5. Thus, if andi calculates a *local* substitution rate above 0.5 for the framed section of two anchors, it is either an exceptionally divergent region or an error. So far andi treats them as highly divergent. However it might be better to simply exclude these regions, or to compute a local alignment which may lead to more accuracy. Again, the results with respect to accuracy and performance need to be evaluated.

7.2 Performance

Sometimes, the efficiency of a tool limits its effectiveness; this is the case for alignments. Their slow performance limits the data to either short sequences or a few long sequences. Computing a multiple sequence alignment or even all pairwise alignments for 3085 *S. pneumoniae* genomes, is simply unfeasible. Here a more efficient tool can have increased effectiveness.

Figure 7.1 shows the performance of andi on the PNEU3085 data set over different versions. The big drop in runtime (drawn through line) from version 0.5 to 0.7 is the result of caching (see Section 5.6).[2] A 16% improvement was achieved in version 0.8 by the use of the FVC array (Section 6.6). The upcoming version 0.9 will use child arrays and thus, gain another 24% at even further memory reduction to just $9\,\mathrm{GB}$.[3] This performance is unmatched by any other publicized method for distance estimation.

Even though the performance of andi is already quite good, it can still be improved. Using an ESA, the longest match problem (see Definition 12) can be solved in time $O(m \cdot |\Sigma|)$. With a *suffix tray* (a joined data structure of a suffix tree and a suffix array) the same problem can be solved in $O(m + \log |\Sigma|)$ time [Cole et al., 2014]. However, as the alphabet used

[2] The version 0.6 was a pure test release and hence is not listed here.
[3] All measurements of andi in this thesis were done using v0.8.1 unless stated otherwise.

in andi is small, this theoretical speed up may not be noticeable in practice. Instead, the use of smaller data structures may lead to faster code due to caching effects. The former can be achieved either via careful programming (e. g., merging the FVC into the most significant bits of the LCP) or through compressed data structures. There have been many advances in the field of compressed indexes (for a »quick tour« see [Grossi, 2011]). The state of the art is that a SA (usually $O(n)$ *words*) can be stored in $n + \log \sigma + o(n + \log \sigma)$ *bits* using a compressed suffix array (CSA). It has to be evaluated whether the improvement in memory usage comes at a negligible runtime cost.

A lower limit for the memory usage of andi is the size of the data set. Thus, even with the `low-memory` switch, andi still uses $6.8\,\mathrm{GB}$ for the PNEU3085 case. A trivial method to improve on this, is to compress the data set in RAM. As the genetic alphabet can be represented using just two bits, a four-fold reduction in size is possible. With sophisticated compression algorithms such as bzip2 or xz a higher compression ratio at the cost of runtime could also be achieved.

7.3 psufsort

In the `low-memory` mode of andi it is a necessity for the SACA to run in parallel across multiple processors. As libdivsufsort has poor CPU utilization (see Section 5.5), psufsort was created. In some use cases psufsort was indeed faster than libdivsufsort, but most of the time, it is much slower and needs to be improved. A *repeat detection* will protect the algorithm from showing worst-case behavior and thus, make it much faster. Further optimizations to the implementation can improve the performance: So far, the parallelization is implemented using OpenMP pragmas. This makes coding easy, but can lead to synchronization overhead if the buckets are not filled uniformly. Thus, a custom scheduling mechanism using the concurrency features of C++11 may lead to a significant performance boost.

A Pseudocode

A.1 Improved Two-Stage Algorithm

```
1  fn improved−two−stage
2  requires T, Σ
3
4
5  // Initialize
6  let n ← |T|
7  let σ ← |Σ|
8  let SA ← array[n] of number
9  let Bucket_L ← array[σ] of number
10 let Bucket_S⁻ ← array[σ,σ] of number
11 let Bucket_S* ← array[σ,σ] of number
12
13
14 Bucket_S*[$].size ← 1
15 i ← n−1
16 goto line 29
17
18 // Classify all suffixes
19 while i ≥ 0 do
20     if T[i] ≥ T[i+1] do
21         Bucket_L[T[i]]. size++ // Type L
22         i ← i−1
23         goto line 19
24     end
25
26     Bucket_S*[T[i], T[i+1]]. size++ // Type S*
27     i ← i−1
28
29     while i ≥ 0 and T[i] ≤ T[i+1] do
30         Bucket_S⁻[T[i], T[i+1]]. size++ // Type S⁻
31         i ← i−1
32     end
33 end
34
35 // Correctly handle the empty suffix
36 SA[0] ← n
37
```

```
38 // Calculate the starting point for each bucket
39 let pos ← 0
40 for i=0 to σ do
41     // Type L suffixes are smaller than their
42     // corresponding Type S suffixes (See Lemma 2)
43     Bucket_L[i].start ← pos
44     pos ← pos + Bucket_L[i].size
45
46     for j=0 to σ do
47         Bucket_S*[i,j].start ← pos
48         pos ← pos + Bucket_S*[i,j].size
49
50         Bucket_S⁻[i,j].start ← pos
51         pos ← pos + Bucket_S⁻[i,j].size
52     end
53 end
54
55
56 // Fill the S* buckets
57 let Temp_S* ← Bucket_S* // Create a copy of the Type S* buckets
58 i ← n−1
59 while i ≥ 0 do
60     if T[i] ≥ T[i+1] do
61         i ← i−1
62         goto line 59 // skip Type L
63     end
64
65     SA[ Temp_S*[T[i], T[i+1]].start ] ← i // insert suffix
66     Temp_S*[T[i], T[i+1]].start ← Temp_S*[T[i], T[i+1]].start + 1
67     i ← i−1
68
69     while i ≥ 0 and T[i] ≤ T[i+1] do // skip Type S⁻
70         i ← i−1
71     end
72 end
73
74
75 // Sort the Type S* suffixes
76 for i=0 to σ do
77     for j=0 to σ do
78         let Bucket_begin ← Bucket_S*[i,j].start
79         let Bucket_end ← Bucket_begin + Bucket_S*[i,j].size − 1
80
81         // Call an external multikey sorting routine on the bucket.
82         // The 2 resembles the depth upto with the strings are
83         // already sorted, i.e. two characters. This call can be
84         // done asynchronously.
```

```
85            mksort( Bucket_begin, Bucket_end, 2)
86        end
87    end
88
89
90    // Sort all Type S⁻ suffixes
91    for i=n to 0 do
92        j ← SA[i]
93
94        if j ≠ ⊥ and T[j−1] ≤ T[j] do
95            let B ← Bucket_S⁻[ T[j−1], T[j]]
96            SA[B.start + B.size − 1] ← j−1
97            B.size ← B.size − 1
98        end
99    end
100
101
102   // Sort all Type L suffixes
103   for i=0 to n+1 do
104       j ← SA[i]
105
106       if j ≠ ⊥ and SA[Bucket_L[T[j]].start] ≠ 0 and T[j−1] ≥ T[j] do
107           SA [ Bucket_L[T[j ]]. start ] ← j−1
108           Bucket_L[T[j ]]. start ← Bucket_L[T[j]]. start + 1
109       end
110   end
111
112
113   output SA
```

A.2 FVC Construction

```
1   fn init_FVC
2   requires S, SA, LCP
3
4   FVC[0] ← '\0'
5   for i=1 to |S| do
6       FVC[i] ← S[SA[i] + LCP[i]]
7   end
```

A.3 `get_interval` with Child Arrays

```
1   fn get_interval
2   requires S, SA, LCP, CLD
3   input (l−[i .. j ], m), a
4
5   do
```

```
6        if S[ SA[i] + l ] = a then
7            j ← m−1
8            if LCP[i] <= LCP[m] then
9                m ← CLD[j+1].L
10           else
11               m ← CLD[i].R
12           end
13           goto line 25
14       end
15
16       if m = j then
17           break
18       end
19
20       m ← CLD[m].R
21   while LCP[m] = l // loop over all subintervals
22
23   // final sanity check
24   if S[SA[i] + l ] = a then
25       l ← LCP[m]
26       output (l−[i .. j ], m)
27   else
28       output ⊥
29   end
```

Notation

SA suffix array
SACA suffix array construction algorithm
LCP longest common prefix
ESA enhanced suffix array
CSA compressed suffix array
DNA deoxyribonucleic acid
SNP single nucleotide polymorphism
RMQ range minimum query
FVC first variant character
CPU central processing unit
GPU graphics processing unit
RAM random access memory
PRAM parallel random access memory
CREW concurrent read exclusive write
CLD child array

Bibliography

[Abouelhoda et al., 2002] Abouelhoda, M., Kurtz, S. and Ohlebusch, E. (2002). The Enhanced Suffix Array and Its Applications to Genome Analysis. In Algorithms in Bioinformatics vol. 2452, of Lecture Notes in Computer Science pp. 449–463.

[Abouelhoda et al., 2004] Abouelhoda, M. I., Kurtz, S. and Ohlebusch, E. (2004). Replacing Suffix Trees with Enhanced Suffix Arrays. J. of Discrete Algorithms *2*, 53–86.

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W. and Lipman, D. J. (1990). Basic local alignment search tool. Journal of Molecular Biology *215*, 403–410.

[Angiuoli and Salzberg, 2011] Angiuoli, S. V. and Salzberg, S. L. (2011). Mugsy: fast multiple alignment of closely related whole genomes. Bioinformatics *27*, 334–342.

[Bentley and McIlroy, 1993] Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. Software: Practice and Experience *23*, 1249–1265.

[Campbell and Reece, 2011] Campbell, N. and Reece, J. (2011). Biologie. 8th edition, Pearson.

[Cantone and Faro, 2014] Cantone, D. and Faro, S. (2014). Improved and self-tuned occurrence heuristics. Journal of Discrete Algorithms *28*, 73–84.

[Chapman, 2009] Chapman, A. (2009). Numbers of Living Species in Australia and the World.

[Chewapreecha et al., 2014] Chewapreecha, C., Marttinen, P., Croucher, N. J., Salter, S. J., Harris, S. R., Mather, A. E., Hanage, W. P., Goldblatt, D., Nosten, F. H., Turner, C., Turner, P., Bentley, S. D. and Parkhill, J. (2014). Comprehensive Identification of Single Nucleotide Polymorphisms Associated with Beta-lactam Resistance within Pneumococcal Mosaic Genes. PLoS Genet *10*.

[Cole et al., 2014] Cole, R., Kopelowitz, T. and Lewenstein, M. (2014). Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. Algorithmica *71*, 1–17.

[Cormen et al., 2009] Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009). Introduction to Algorithms. 3rd edition, MIT Press.

[Darwin, 1859] Darwin, C. (1859). The Origin of Species. 150th anniversary edition, Bridge-Logos.

[Daskalakis and Roch, 2013] Daskalakis, C. and Roch, S. (2013). Alignment-free phylogenetic reconstruction: Sample complexity via a branching process analysis. The Annals of Applied Probability *23*, 693–721.

[Deo and Keely, 2013] Deo, M. and Keely, S. (2013). Parallel Suffix Array and Least Common Prefix for the GPU. SIGPLAN Not. *48*, 197–206.

[Domazet-Lošo and Haubold, 2009] Domazet-Lošo, M. and Haubold, B. (2009). Efficient estimation of pairwise distances between genomes. Bioinformatics *25*, 3221–3227.

[Felsenstein, 2004] Felsenstein, J. (2004). Inferring Phylogenies. Sinauer Associates, Inc.

[Felsenstein, 2005] Felsenstein, J. (2005). PHYLIP (Phylogeny Inference Package). Distributed by the author. Department of Genome Sciences, University of Washington.

[Filion, 2015] Filion, G. (accessed 2015). The Grand Locus: If cars were made by bioinformaticians. `http://blog.thegrandlocus.com/2015/01/if-cars-were-made-by-bioinformaticians`.

[Fischer and Heun, 2007] Fischer, J. and Heun, V. (2007). A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In Proc. ESCAPE. LNCS pp. 459–470, Springer.

[Free Software Foundation, 2007] Free Software Foundation (2007). Gnu General Public License. `https://gnu.org/licenses/gpl.html`.

[Grossi, 2011] Grossi, R. (2011). A Quick Tour on Suffix Arrays and Compressed Suffix Arrays. Theor. Comput. Sci. *412*, 2964–2973.

[Haubold, 2014] Haubold, B. (2014). Alignment-free phylogenetics and population genetics. Briefings in Bioinformatics *15*, 407–418.

[Haubold et al., 2014] Haubold, B., Klötzl, F. and Pfaffelhuber, P. (2014). andi: Fast and accurate estimation of evolutionary distances between closely related genomes. Bioinformatics .

[Haubold et al., 2009] Haubold, B., Pfaffelhuber, P. and Domazet-Lošo, M. (2009). Estimating Mutation Distances from Unaligned Genomes. Journal of Computational Biology *16*, 1487–1500.

[Hudson, 2002] Hudson, R. R. (2002). Generating samples under a Wright–Fisher neutral model of genetic variation. Bioinformatics *18*, 337–338.

[Itoh and Tanaka, 1999] Itoh, H. and Tanaka, H. (1999). An Efficient Method for in Memory Construction of Suffix Arrays. In Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware SPIRE '99 pp. 81–, IEEE Computer Society.

[Jukes and Cantor, 1969] Jukes, T. H. and Cantor, C. R. (1969). Evolution of protein molecules. Mammalian protein metabolism *3*, 21–132.

[Kasai et al., 2001] Kasai, T., Lee, G., Arimura, H., Arikawa, S. and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In Combinatorial Pattern Matching pp. 181–192, Springer.

[Ko and Aluru, 2003] Ko, P. and Aluru, S. (2003). Space efficient linear time construction of suffix arrays. In Journal of Discrete Algorithms pp. 200–210, Springer.

[Kuhner and Felsenstein, 1994] Kuhner, M. K. and Felsenstein, J. (1994). A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. Molecular Biology and Evolution *11*, 459–468.

[Kulla and Sanders, 2006] Kulla, F. and Sanders, P. (2006). Scalable Parallel Suffix Array Construction. In Recent Advances in Parallel Virtual Machine and Message Passing Interface vol. 4192, of Lecture Notes in Computer Science pp. 22–29.

[Kurtz, 2014] Kurtz, S. (accessed 2014). vmatch. `http://vmatch.de/`.

[Kärkkäinen and Sanders, 2003] Kärkkäinen, J. and Sanders, P. (2003). Simple Linear Work Suffix Array Construction.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In Code Generation and Optimizatio pp. 75–88, International Symposium on Code Generation and Optimization.

[Leimeister et al., 2014] Leimeister, C.-A., Boden, M., Horwege, S., Lindner, S. and Morgenstern, B. (2014). Fast alignment-free sequence comparison using spaced-word frequencies. Bioinformatics *30*, 1991–1999.

[Manber and Myers, 1990] Manber, U. and Myers, G. (1990). Suffix-Arrays: A New Method for On-Line String Searches.

[Maniscalco, 2015] Maniscalco, M. (accessed Jan. 2015). The Gauntlet Corpus. `http://www.michael-maniscalco.com/index.htm#ID_GUANTLET`.

[Maniscalco and Puglisi, 2006] Maniscalco, M. A. and Puglisi, S. J. (2006). Faster lightweight suffix array construction. In 17th Australasian Workshop on Combinatorial Algorithms pp. 16–29, Univ. Ballavat.

[Manzini, 2004] Manzini, G. (2004). Two space saving tricks for linear time LCP computation. In Proc. SWAT. Volume 3111 of Lecture Notes in Computer Science pp. 372–383, Springer.

[Manzini and Ferragina, 2004] Manzini, G. and Ferragina, P. (2004). Engineering a Lightweight Suffix Array Construction Algorithm. Algorithmica *40*, 33–50.

[Marçais and Kingsford, 2011] Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics *27*, 764–770.

[Maurer-Stroh et al., 2013] Maurer-Stroh, S., Gunalan, V., Wong, W. and Eisenhaber, F. (2013). A Simple Shortcut to Unsupervised Alignment-Free phylogenetic genome Groupings, even from Unassembled sequencing reads. J. Bioinformatics and Computational Biology *11*.

[Morgan et al., 1915] Morgan, T., Sturtevant, A., Muller, H. and Bridges, C. (1915). The Mechanism of Mendelian Heredity. Henry Holt and Company.

[Morgenstern et al., 2015] Morgenstern, B., Zhu, B., Horwege, S. and Leimeister, C. (2015). Estimating evolutionary distances between genomic sequences from spaced-word matches. Algorithms for Molecular Biology *10*, 5.

[Morgenstern et al., 2014] Morgenstern, B., Zhu, B., Horwege, S. and Leimeister, C.-A. (2014). Estimating Evolutionary Distances from Spaced-Word Matches. In Algorithms in Bioinformatics vol. 8701, of Lecture Notes in Computer Science pp. 161–173.

[Mori, 2005] Mori, Y. (2005). Short description of improved two-stage suffix sorting algorithm. `http://homepage3.nifty.com/wpage/software/itssort.txt`.

[Musser, 1997] Musser, D. R. (1997). Introspective Sorting and Selection Algorithms. Software: Practice and Experience *27*, 983–993.

[Mäusle, 2012] Mäusle, S. (2012). Implementation and Comparison of Algorithms for Constructing and Visualizing Phylogenetic Trees. Bachelor's thesis University of Lübeck.

[Navarro et al., 1997] Navarro, G., Kitajima, J., Ribeiro-Neto, B. and Ziviani, N. (1997). Distributed generation of suffix arrays. In Combinatorial Pattern Matching vol. 1264, of Lecture Notes in Computer Science pp. 102–115.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. SIGPLAN Not. *42*, 89–100.

[Newton et al., 2010] Newton, R. J., Griffin, L. E., Bowles, K. M., Meile, C., Gifford, S., Givens, C. E., Howard, E. C., King, E., Oakley, C. A., Reisch, C. R., Rinta-Kanto, J. M., Sharma, S., Sun, S., Varaljay, V., Vila-Costa, M., Westrich, J. R. and Moran, M. A. (2010). Genome characteristics of a generalist marine bacterial lineage. The ISME Journal *4*, 784–798.

[Ohlebusch, 2013] Ohlebusch, E. (2013). Bioinformatics Algorithms. 1st edition, Oldenbusch Verlag.

[Osipov, 2012] Osipov, V. (2012). Parallel Suffix Array Construction for Shared Memory Architectures. In Proceedings of the 19th International Conference on String Processing and Information Retrieval SPIRE'12 pp. 379–384, Springer.

[Puglisi et al., 2007] Puglisi, S., Smyth, M. and Turpin, A. (2007). A Taxonomy of Suffix Array Construction Algorithms. ACM Computing Surveys *39*.

[Rajasekaran and Nicolae, 2014] Rajasekaran, S. and Nicolae, M. (2014). An elegant algorithm for the construction of suffix arrays. Journal of Discrete Algorithms *27*, 21–28.

[Rambaut, 2015] Rambaut, A. (accessed 2015). FigTree. `http://tree.bio.ed.ac.uk/software/figtree/`.

[Robinson and Foulds, 1981] Robinson, D. and Foulds, L. (1981). Comparison of phylogenetic trees. Mathematical Biosciences *53*, 131–147.

[Schürmann, 2007] Schürmann, K.-B. (2007). Suffix Arrays in Theory and Practice. Dr. rer. nat. Bielefeld University.

[Shun et al., 2012] Shun, J., Blelloch, G. E., Fineman, J. T., Gibbons, P. B., Kyrola, A., Simhadri, H. V. and Tangwongsan, K. (2012). Brief Announcement: The Problem Based Benchmark Suite. In Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '12 pp. 68–70, ACM.

[Tang et al., 2014] Tang, J., Hua, K., Chen, M., Zhang, R. and Xie, X. (2014). A novel k-word relative measure for sequence comparison. Computational Biology and Chemistry *53, Part B*, 331–338.

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

[Whidden et al., 2013] Whidden, C., Beiko, R. G. and Zeh, N. (2013). Fixed-Parameter Algorithms for Maximum Agreement Forests. SIAM Journal on Computing *42*, 1431–1466.

[Yi and Jin, 2013] Yi, H. and Jin, L. (2013). Co-phylog: an assembly-free phylogenomic approach for closely related organisms. Nucleic Acids Research *41*.

[Zuckerkandl and Pauling, 1962] Zuckerkandl, E. and Pauling, L. (1962). Molecular disease, evolution, and genic heterogeneity. Horizons in Biochemistry