kmacs: the \textit{k}-Mismatch Average Common Substring
Approach to alignment-free sequence comparison

Chris-Andre Leimeister\textsuperscript{1,}\textsuperscript{*} and Burkhard Morgenstern\textsuperscript{1,2}
\textsuperscript{1}University of Göttingen, Institute of Microbiology and Genetics, Department of Bioinformatics, Goldschmidtstr. 1, 37073 Göttingen, Germany, and \textsuperscript{2}Université d’Évry Val d’Essonne, Laboratoire Statistique et Génome, UMR CNRS 8071, USC INRA, 23 Boulevard de France, 91037 Évry, France

ABSTRACT

Motivation: Alignment-based methods for sequence analysis have various limitations if large data sets are to be analysed. Therefore, alignment-free approaches have become popular in recent years. One of the best known alignment-free methods is the \textit{average common substring (ACS) approach} that defines a distance measure on sequences based on the average length of longest common words between them. Herein, we generalize this approach by considering longest common substrings with \textit{k} mismatches. We present a greedy heuristic to approximate the length of such \textit{k}-mismatch substrings and we describe \textit{kmacs}, an efficient implementation of this idea based on generalized enhanced suffix arrays.

Results: To evaluate the performance of our approach, we applied it to phylogeny reconstruction using a large number of DNA and protein sequence sets. In most cases, phylogenetic trees calculated with \textit{kmacs} were more accurate than trees produced with established alignment-free methods which are based on exact word matches. Especially on protein families, our method seems to be superior. On simulated protein families, \textit{kmacs} even outperformed a classical approach to phylogeny reconstruction using multiple alignment and maximum likelihood.

Availability: \textit{kmacs} is implemented in \texttt{c++} and the source code is freely available at: http://kmacs.gobics.de/

Contact: chris.leimeister@stud.uni-goettingen.de

1 INTRODUCTION

Comparative sequence analysis traditionally relies on pairwise or multiple sequence alignment. With the huge data sets that are produced by next-generation sequencing technologies, however, today’s alignment algorithms reach their limits. Thus, with the growing number of completely or partially sequenced genomes, there is an urgent demand for faster sequence-comparison methods. Over the past two decades, a wide variety of alignment-free approaches were proposed (Vinga and Almeida, 2003). While aligning two sequences takes time proportional to the product of their lengths, most alignment-free methods run in linear time. They are therefore increasingly used for genome-based phylogeny reconstruction and for large-scale protein sequence comparison. It is known, however, that alignment-free methods are generally less accurate than alignment-based approaches.

Most alignment-free methods calculate the relative frequencies of words of a fixed length \textit{k}, also called \textit{k}-mers. Other methods are based on variable-length matches, they have the advantage that it is not necessary to specify a fixed word length (Didier et al., 2012; Comin and Verzotto, 2012). These programs achieve usually better results than approaches relying on a fixed word length. However, algorithms using variable word lengths are typically more complex and require more sophisticated data structures than methods relying on fixed word lengths.

A well-known approach that uses word matches of variable length is the \textit{average common substring (ACS) method} (Ulitsky et al., 2006) which calculates for each position \textit{i} in one sequence the length of the longest substring starting at \textit{i} and matching some substring of a second sequence. As a further development of this idea, the \textit{shortest unique substring (shustring)} approach has been proposed by Haubold et al. (2005). These authors also derived an estimator for the number of substitutions per site between two unaligned sequences based on the average \textit{shustring} length; they implemented this approach in the program \textit{K2} (Haubold et al., 2009). ACS and \textit{shustrings} can be calculated efficiently using \textit{suffix trees} (Weiner, 1973).

As the above mentioned methods, most approaches for alignment-free phylogeny reconstruction are based on \textit{exact} word matches. Recently, we suggested to use \textit{spaced-k-mers} defined by pre-defined patterns of \textit{match} and \textit{don’t care} symbols, instead of contiguous \textit{k}-mers (Boden et al., 2013; Leimeister et al., 2014). The aim of the present study is to apply the idea of \textit{inexact} matches to word matches of varying lengths. We generalize the ACS approach by considering, for each position \textit{i} in one sequence, the longest substring starting at \textit{i} and matching some substring in the second sequence with \textit{k} mismatches. We propose an efficient heuristic to approximate the lengths of these substrings, and we describe \textit{kmacs}, an implementation of this approach based on \textit{generalized enhanced suffix arrays}. A web server for our program is described in Horwege et al. (2014).

\textsuperscript{*}To whom correspondence should be addressed
2 APPROACH

2.1 The Average Common Substring Approach and k-mismatch substrings

As usual, for a sequence S over an alphabet Σ, S[i] is the i-th element of S, by |S| we denote the length of S and S[i..j] is the (contiguous) substring of S from i to j. In particular, S[i..|S|] is the i-th suffix of S. For two sequences S_1 and S_2, the ACS approach determines for every position i in S_1 the length s_1(i) of the longest substring of S_1 starting at position i and exactly matching some substring in S_2. The lengths s_1(i) are averaged and normalized to define a similarity measure

\[
L(S_1, S_2) = \frac{1}{|S_1|} \sum_{i=1}^{|S_1|} s_1(i)
\]

which is turned into a (non-symmetric) distance measure by defining

\[
d(S_1, S_2) = \frac{\log(|S_2|)}{L(S_1, S_2)} - \frac{\log(|S_1|)}{L(S_1, S_1)}
\]

To obtain a symmetric distance, the distance between S_1 and S_2 is then defined by Ulitsky et al. (2006) as

\[
d_{ACS}(S_1, S_2) = \frac{d(S_1, S_2) + d(S_2, S_1)}{2}
\]

In this paper, we generalize this distance measure by using substring matches with k mismatches instead of exact matches. That is, instead of using the maximum substring lengths s_1(i), we define s_k(i) as the length of the longest substring of S_1 starting at position i and matching some substring of S_2 with up to k mismatches, minus k. (We subtract k from the length of this string, counting only the matching positions). s_k(i) is defined accordingly. We then define a distance measure as above, but with s_k(i) replaced by s_k(i). Note that in the special case k = 0, we have s_0(i) = s_1(i), so in this case our distance is exactly the distance d_{ACS}.

2.2 Approximating the Length of k-Mismatch Substrings

For a pair of sequences, the exact values s_k(i) can be calculated in O(k * n^2) time using suffix trees or similar data structures where n is the maximal length of the sequences. Since we want to compare sequences in linear time, however, we propose a heuristic to approximate these values. To do so, we first calculate for each position i in S_1 the length s_1(i) of the longest common substring starting at i matching a substring of S_2, as is done in ACS. Let j be the start of this matching substring in S_2; the character S_1[i + s(i)] must therefore differ from S_2[j + s(i)]. We then extend this match without gaps i from position i + s(i) + 1 and in S_2 from j + s(i) + 1, until the next mismatch occurs. This is repeated until the k + 1-th mismatch or the end of one of the two sequences is reached.

In the example below, for position i = 4 in S_1 and with k = 2 mismatches, our approach would return the following k-mismatch common substring, starting at position j = 2 in S_2:

<table>
<thead>
<tr>
<th>S_1</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_2</td>
<td>A</td>
<td>T</td>
<td>G</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>T</td>
</tr>
</tbody>
</table>

To obtain this k-mismatch common substring, our program would first determine the longest common substring for position i = 4 in S_1 that exactly matches a substring in S_2. We find such a match at position j = 2 in S_2 with the length s_1(4) = 2. Then this match is extended without gaps until the third mismatch is reached. The length of this 2-mismatch substring is 7, so we have s_2(4) = 5 (note that in the definition of s_k(i), we count only the matching positions).

It should be mentioned that, for a position i in S_1, the corresponding position j in S_2 of the longest exact match to a substring starting at i may not be unique. Consider, e.g. position i = 2 in the first sequence of the above example:

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

Here, the substring AT starting at position 2 in S_1 is the longest substring starting at this position and matching a substring of S_2 — but this substring occurs at positions 1, 5 and 10 in S_2. In such a case we calculate all k-mismatch extensions of these occurrences as described above, and we define s_k(i) as length of the maximal possible extension minus k.

The above heuristic reduces the complexity of finding the k mismatch maximal substring lengths from O(k * n^2) to O(k * n + z) where z is the average number of maximal matches to a substring in S_2 starting at a position i in S_1. In principle, this complexity could be achieved by using suffix trees (Weiner, 1973) as the underlying data structure. Here, one would build a generalized suffix tree for the sequences in O(|S_1| + |S_2|) time, e.g. using Ukkonen’s algorithm (Ukkonen, 1995). To determine the longest substring starting at i in S_1 and also occurring in S_2, one needs to find the lowest node v in the suffix tree that is above leaf i and also above some leaf that belongs to S_2. The length s_k(i) of the longest common substring starting at i is then the string depth of the node v, that is the length of the edge labels on the path from the root to v. Moreover, the leaves below v appertaining to S_2 exactly correspond to the positions of this longest exact match in S_2.

Next, we want to extend the longest exact matches that we have found by this procedure until the k + 1-th mismatch is found. Thus, we need to be able to find the longest exact match between two sequences starting at two given positions i and j (the positions after a mismatch, in our case). In a suffix-tree approach, this could be accomplished by lowest common ancestor (LCA) queries. Similar to the above mentioned approach, we would have to look up the lowest node v that is above both leaves i and j; the string depth of v is then the length of the longest exact match starting at i and j, respectively. LCA queries can be carried out for any i and j in constant time after a linear-time preprocessing step (Harel and Tarjan, 1984), resulting in k constant-time LCA queries for the full k-mismatch extension of an exact longest match.

3 IMPLEMENTATION

Abouelhoda et al. (2004) have shown that every algorithm that uses suffix-trees can be replaced by an algorithm using enhanced suffix arrays that has the same complexity. Here, an enhanced suffix array is defined as a data structure ‘consisting of the suffix array and additional tables’. Both, suffix trees and enhanced suffix arrays, can be calculated in linear time and space, but suffix arrays require substantially less memory per input character than suffix trees do (Manber
and Myers, 1990). In our implementation, we therefore used enhanced suffix arrays instead of suffix trees, making use of recent improvements of linear-time suffix array construction algorithms.

A suffix array $SA$ of a string $S = S[1] \ldots S[n]$ is a permutation of the indices $1 \ldots n$ according to the lexicographical ordering of the corresponding suffixes. That is, we have $SA[i] = j$ if the $j$-th suffix of $S$ is at the $i$-th position in the lexicographical ordering of all suffixes of $S$. In addition to the suffix array $SA$, we need the so-called longest common prefix (LCP) array for $S$. Here, the entry $LCP[i]$ stores the length of the longest common prefix of the $SA[i]$-th suffix and its predecessor in $SA$, the $SA[i-1]$-th suffix. The suffix array $SA$ of a sequence $S$ together with the corresponding LCP array is called, in this context, the enhanced suffix array of $S$. To calculate enhanced suffix arrays of $S$, we used a program described by Fischer (2011) which is available at http://algo2.iti.kit.edu/english/1828.php. The underlying algorithm is based on sais-literate by Yuta Mori, a fast implementation of induced sorting (Nong et al., 2009). Suffix arrays provide an efficient solution to our longest k-mismatch substring problem.

For a single sequence $S$ and a position $SA[i]$ in $S$, the enhanced suffix array of $S$ can be used to find the length of the longest substring in $S$ starting at a different position in $S$ and matching a substring starting at $SA[i]$. It is easy to see that this substring must be the longest common prefix of the $SA[i]$-th suffix with one of its neighbours in $S$, i.e. either with the $SA[i+1]$-th or the $SA[i-1]$-th suffix, whichever is longer. With an enhanced suffix array, the length of this substring can be obtained as the maximum of the values $LCP[i]$ and $LCP[i+1]$ and can therefore be looked up in constant time. The position where this second substring starts is then either $SA[i-1]$ or $SA[i+1]$ – or both of these positions –, depending on where the maximum is reached.

If matches between two sequences are to be found, the situation is slightly more complicated. For a position in sequence $S_1$, we want to find a position in $S_2$ such that the common substring starting at these two positions is maximal, and vice versa. To solve this problem, we build the generalized enhanced suffix array of our sequences, i.e. the enhanced suffix array of the concatenated sequence $S := S_1 \$ S_2$ where $\$ is a special character not contained the alphabet $S$; see also Babenko and Starikovskaya (2008) for a related approach. Thus, each suffix from $S_1$ or $S_2$ is represented in lexicographical order by an entry in $SA$. Fig. 1 shows the enhanced suffix array for two sequences.

To find the length of the longest substring starting at $SA[i]$ in one sequence, matching a substring of the other sequence, and its occurrences there, we need to look up the longest integer $p_1(i)$ with $p_1(i) < i$, such that $SA[p_1(i)]$ belongs to the other sequence. Correspondingly, we need the smallest integer $p_2(i)$ with $p_2(i) > i$ with $SA[p_2(i)]$ from to the other sequence. The length of this common substring is then given as the minimum of all $LCP$ values between $p_1(i) + 1$ and $i$ or the minimum between the $LCP$ values between $i + 1$ and $p_2(i)$ – whichever minimum is larger. Formally, the length of the longest substring starting at a position $SA[i]$ and matching a substring of the respective other sequence is given as

$$s(SA[i]) = \min_{p_1(i) < x \leq i} \min_{i < y \leq p_2(i)} LCP[x, y]$$

with $p_1$ and $p_2$ defined as above.

The position of this longest substring in $S$ is then $SA[p_1(i)]$ or $SA[p_2(i)]$ (or both), depending on where the maximum in equation (4) is reached. Note that all positions in this formula refer to the concatenated sequence $S$, but it is trivial to retrieve the positions in the original sequences $S_1$ and $S_2$ from these values by subtracting $|S_1| + 1$ where necessary.

As an example, consider Fig 1. For $i = 6$, we want to find the longest common substring starting at $SA[6] = 10$ (marked by an arrow) that exactly matches a substring starting at some position in the other sequence. Position $SA[6] = 10$ in the concatenated sequence $S$ corresponds to a position in sequence $S_2$, so we have $p_1(6) = 4$, as 4 is the largest integer smaller than 6 such that $SA[4]$ belongs to the other sequence, i.e. to $S_1$. Similarly, we obtain $p_2(6) = 8$. According to equation (4), we get

$$s(SA[6]) = \max \{ \min\{5, 3\}, \min\{1, 0\} \} = \max\{3, 0\} = 3.$$

Position 10 in $S$ corresponds to position 3 in the original sequence $S_2$ so, as a result, we obtain $s_2(3) = 3$, i.e. the longest substring starting at position 3 in $S_2$ matching a substring from $S_1$ has length 3 (the substring itself is “anan”).

All values $s(i)$ can be calculated for the entire concatenated string $S$ in linear time using Algorithm 1. Here, the first loop computes $\min_{p_1 \leq i \leq x} LCP[x]$ for all indices $i$ and stores them as $s[i]$. Then the second loop calculates $\min_{y \leq p_2 \leq \max} LCP[y]$ and updates $s[i]$ if the result is greater than the actual value of $s[i]$. This way, Algorithm 1 applies equation (4) for to all indices $i$ and stores the corresponding values $s[i]$. Finally, for our heuristic we need to find for an index $i$ all positions belonging to the respective other sequence, where a match of length $s(i)$ occurs. This can be achieved by a simple extension of Algorithm 1. Without loss of generality, we assume that the first

$$i \quad SA[i] \quad Suffix \quad LCP[i]$$

| 1 | 7 | $\$ananas | - |
| 2 | 6 | aS$anananas$ | 0 |
| 3 | 4 | ana$anananas$ | 1 |
| 4 | 2 | anana$anananas$ | 3 |
| 5 | 8 | ananas | 5 |
| 6 | 10 | anas | 3 |
| 7 | 12 | as | 1 |
| 8 | 1 | banana$anananas$ | 0 |
| 9 | 5 | na$anananas$ | 0 |
| 10 | 3 | nana$anananas$ | 2 |
| 11 | 9 | nana | 4 |
| 12 | 11 | nas | 2 |
| 13 | 13 | $s$ | 0 |

Fig. 1. Generalized SA and LCP array for the strings $S_1 = banana$ and $S_2 = ananas$, concatenated by the symbol $. Suffices of $S_1$ with $S_2$ starting in $S_1$ are shown in orange, suffices starting in $S_2$ are in blue.
Algorithm 1 Calculation of equation (4)

Require: SA [generalized suffix array for S₁ and S₂ of length n]
Require: LCP [corresponding longest common prefix array]
Ensure: s [stores the the results of equation (4)]

\[
\text{min} \leftarrow 0
\]

for \(i = 2\) to \(n - 1\) do
  if \(SA[i] \text{ and } SA[i + 1]\) belong to the same sequence then
    if \(LCP[i + 1] < \text{min}\) then
      \(\text{min} \leftarrow LCP[i + 1]\)
    end if
    \(s[i + 1] \leftarrow \text{min}\)
  else
    \(\text{min} \leftarrow LCP[i + 1]\)
  end if
end for

\[
\text{min} \leftarrow 0
\]

for \(i = n\) to \(2\) do
  if \(SA[i]\) and \(SA[i + 1]\) belong to the same sequence then
    if \(LCP[i] < \text{min}\) then
      \(\text{min} \leftarrow LCP[i]\)
    end if
    \(s[i - 1] \leftarrow \max(\text{min}, s[i - 1])\)
  else
    \(\text{min} \leftarrow LCP[i]\)
    \(s[i - 1] \leftarrow \max(\text{min}, s[i - 1])\)
  end if
end for

minimum in equation (4) is strictly larger than the second minimum, so \(p_1(i)\) is a position where a maximal match to the other sequence occurs (as was the case in our small example above). To find possible additional matching positions, we consider all indices \(p \leq p_1(i)\) in descending order, as long as one has

\[
LCP[p + 1] \leq \min_{p_1(i) \leq k \leq i} LCP[k]
\]

For all such \(p\) that belong to the other sequence, the positions \(SA[p]\) are occurrences of longest substrings matching a substring starting at \(i\). In our example, we find one further position \(p = 3\), so \(SA[3] = 4\) is an additional occurrence. If the maximum in (4) is achieved by the second term, one proceeds accordingly.

Next, the second step in our approach involves finding the length of the longest common substring starting at pre-defined positions in \(S_1\) and \(S_2\), respectively. Using the enhanced suffix array of a sequence \(S\), the length of the longest substring starting at positions \(SA[i]\) and \(SA[j]\) \((SA[i] < SA[j])\) is given as the minimum over the values \(LCP[p], i < p < j\). There is an approach similar to LCA queries to obtain this value known as range minimum queries (RMQ). A RMQ returns the index of an array \(A\) that stores the smallest element between two specified indices \(l\) and \(r\), denoted as \(\text{RMQ}(A, l, r)\).

Several algorithms are available which can solve RMQ in constant time, after a linear preprocessing step, e.g. Fischer and Heun (2006). The longest common substring starting at \(i\) and \(j\) can be calculated as \(LCP[\text{RMQ}(SA^{-1}[i + 1], SA^{-1}[j])]\) where \(SA^{-1}\) is the inverse suffix array. As a result, the same complexity as for suffix trees can be achieved by using enhanced suffix arrays. In our implementation however, we extend the substrings by matching single characters since in our test runs this ‘naive’ approach was faster than the RMQ implementation that we tested. Nevertheless, our downloadable program has an option for using the RMQ algorithm so the user can compare these two approaches.

4 BENCHMARKING

4.1 Benchmark sequences

To evaluate kmacs and to compare it to other methods of sequence comparison, we applied these methods phylogeny reconstruction. We used a large number of DNA and protein sequence sets for which reliable phylogenetic trees are available, and we measured how similar the constructed trees are to the respective reference trees. The following sequence sets were used in our study:

For eukaryotic DNA comparison, we used a set of 27 primate mitochondrial genomes which were previously used by Haubold et al. (2009) as benchmark for alignment-free methods. These sequences have a total length of 446kb. A benchmark tree is available that is based on multiple alignments.

As prokaryotic genomes, we used a set of 32 Roseobacter genomes which were previously analysed by Newton et al. (2010). They constructed a phylogenetic tree for these sequences based on alignments of 70 universal single-copy genes which we used as reference tree in our study. The total size of this sequence set is 135.9mb.

As benchmark proteins, we used 218 sequence sets contained in the BAliBASE (v3.0) database (Thompson et al., 2005). To obtain reference trees, we applied Maximum Likelihood (Felsenstein, 1981), implemented in the program proml from PHYLIP to the reference multiple alignments in BAliBASE. Since these reference alignments are considered to be reliable, the resulting trees should also be reliable.

In addition to these real-world sequences, we used the program Rose (Stoye et al., 1998) to generate simulated DNA and protein families. Rose generates sets of related sequences based on a probabilistic model of substitutions and insertions/deletions for which the parameters can be adjusted by the user. These sequences are created along a randomly generated tree, starting from one common ancestral sequence at the root of the tree. This way, the ‘evolution’ of the generated sequences is logged, so a reference tree is generated alongside the sequences. We used Rose with default parameters, except for the parameter relatedness which defines the average evolutionary distance between the generated sequences, measured in PAM units. We generated 20 DNA sequence sets, each of which containing 50 sequences with an average length of 16,000 nt using a relatedness value of 70. Further more,
we generated 20 protein sequence sets, each containing 125 sequences with an average length of 300 aa. Here, we set the relatedness to 480.

### 4.2 Compared methods

We compared our new method to seven state-of-the-art alignment-free methods, namely ACS (Ulitsky et al., 2006), $K_r$, v2.0.2 (Haugboll et al., 2009), FFP (Sims et al., 2009), spaced-words (Leimeister et al., 2014), CVTree (Qi et al., 2004), the underlying approach (UA) (Comin and Verzotto, 2012) as well as to a generic $k$-mer-frequency approach. As an eighth method, we ran Clustal W (Thompson et al., 1994) on those sequence sets where this was possible and meaningful. For ACS and the $k$-mer approach, we used our own implementations, namely kmacs with $k = 0$ and our spaced words approach without don’t care positions in the underlying patterns, respectively.

FFP, $K_r$, and CVTree return pairwise distances between the input sequences. For ACS, we calculated distances as defined in (3), and for spaced words and the $k$-mer approach we used the Jensen-Shannon divergence (Lin, 1991), applied to (spaced)-word frequency vectors as explained explained in Leimeister et al. (2014). For each of the five groups of benchmark data, we used the word length $k$ for which the $k$-mer approach produced the best results, i.e., trees with minimal average Robinson-Foulds distances to the reference trees. For spaced words, we used the same value for $k$, even though better results might be possible with different values. Accordingly, on every group of benchmark data, we tested FFP, CVTree and UA with different parameter values and used those which produced the best results on this group.

We then constructed phylogenetic trees by applying Neighbor joining (Saitou and Nei, 1987) to the distance matrices obtained with the different alignment-free methods. Finally, we calculated phylogenetic trees for all sequence sets by applying Maximum Likelihood (Felsenstein, 1981) to the Clustal W multiple alignments. All resulting tree topologies were compared to the topologies of the respective reference trees using the Robinson-Foulds metric (Robinson and Foulds, 1981). For Neighbor joining and to calculate the Robinson-Foulds distances, we used the programs neighbor and treeclust contained in the PHYLIP package (Felsenstein, 1989).

### 5 RESULTS AND DISCUSSION

Figures 2 and 4 – 7 summarize our test results on the five groups of benchmark sequence sets that we used. The plots show the average Robinson-Foulds distances between the produced trees and the corresponding reference trees. Note that for kmacs, results are shown for various values of $k$. For FFP, CVTree, UA and the $k$-mer method, we also used a range of parameter values, but for each of these methods, the figures show only the the best results on the respective group of benchmark sequences. Thus, for a fair comparison, these methods should be compared to the best results of kmacs in the corresponding figure. On the other hand, $K_r$, ACS and Clustal could be used with default parameters which is clearly an advantage of these methods.

Figure 2 contains the test results on the primate mitochondrial genomes. The best method on this data set was our previously developed spaced-words approach; the tree topology produced by this method precisely coincides with the topology of the reference tree, i.e. the Robinson-Foulds distance is zero. The second best methods were FFP and kmacs with $k = 3, 4$ and $64 \leq k \leq 117$. ACS, CVTree, UA, and kmacs with other values for $k$ and $K_r$ performed worse on these data. As an example, Figure 3 compares the tree
calculated with kmacs ($k = 70$) to the alignment-based reference tree from Haubold et al. (2009). The tree topology calculated by kmacs almost coincides with the topology of the reference tree, the RF distance between these trees is 2.

On the Roseobacter genomes, the best methods were kmacs with $k = 4$ and 6, FFP, and CVTree as shown in Figure 4. Spaced words and the generic k-mer approach performed slightly worse. None of the tested methods was able to exactly reconstruct the topology of the reference tree. UA is missing in this comparison, since this program is too slow to be run on full bacterial genomes in reasonable time. For our simulated DNA sequence sets, the results were similar as for the primate mitochondrial genomes, see Figure 5. Here too, spaced-words was the best alignment-free method, followed by kmacs. This time kmacs outperformed the established alignment-free approaches for all values of $k$ that we tested. On our simulated DNA sequences, we could also run a classical approach to phylogeny reconstruction using Clustal W and Maximum Likelihood. Not surprisingly, this slow and accurate method performed better than all alignment-free approaches.

Figure 6 shows the results for the BAliBASE protein sequences. spaced words and kmacs again produced better results than the existing alignment-free methods that we evaluated. This time, there was a large range of values for $k$ where kmacs performed similar or even slightly better than spaced words and both methods outperformed the other alignment-free methods that we tested. As with the previous data set, the classical approach based on multiple sequence alignment performed best; this time the difference between alignment-based and alignment-free methods was larger. This may be due to the fact that multiple-alignment programs are often tuned to perform well on BAliBASE, the main database to evaluate multiple-alignment methods.

Finally, the results on our simulated protein sequences are shown in figure 7. As in most previous examples, spaced words and kmacs outperformed other alignment-free approaches and, as on BAliBASE, kmacs was slightly better than spaced words if $k$ was sufficiently large. Surprisingly, on these benchmark sequences spaced words and kmacs even outperformed Clustal W and Maximum Likelihood, although not dramatically.

So far, we evaluated alignment-free and alignment-based methods indirectly, by applying them to phylogeny reconstruction and comparing the resulting trees to trusted reference trees using the Robinson-Foulds (RF) metric. This is a common procedure to evaluate alignment-free methods. RF distances to reference trees are only a rough measure of accuracy, though, since they are based on tree topologies alone and do not take branch lengths into account. Furthermore, the constructed trees depend not only on the underlying methods for sequence comparison, but also on the methods used for tree reconstruction. A more direct and accurate way of comparing alignment-free methods is to directly compare the distance values that they calculate. This can be done, for example, by plotting the distances produced for simulated sequences against their real evolutionary distances (Haubold et al., 2009). Ideally, this should be a linear relation. Figure 8 shows such plots for the algorithms that we compared in our study.

Table 1 and Table 2 summarize the run times of the different methods that we tested. When used with moderate values of $k$, kmacs is faster than spaced words run with a set of 100 different patterns. $K_r$ was more than one order of magnitude faster than kmacs and spaced words, respectively, while UA was the slowest method. The fastest method was our implementation of the generic word-frequency approach, followed by $K_r$ and CVTree. In general, spaced words used with the single-pattern option is only slightly slower than the k-mer approach. As shown in our companion paper, however, spaced words produces considerably better results when used with multiple patterns (Leimeister et al., 2014). We therefore applied only the multiple-pattern version in the present study.

The relatively long runtime of UA is partially due to the fact that this program is written in Java while all other programs that we tested are written in C++. As expected, the multiple-alignment approaches Clustal W and Clustal Ω (Sievers et al., 2011) were
far slower than the alignment-free methods; the difference in speed between alignment-based and alignment-free methods was between three and four orders of magnitude. All test runs were done on an Intel Core i7 4820k which we overclocked to 4.5ghz.

As explained in section 2.2, kmacs searches for each position $i$ in one sequence the maximum substring starting at $i$ that matches a substring in the second sequence. There can be more than one such maximal match, and all these matches are extended to $k$-mismatch common substrings. Thus, the runtime of kmacs depends on $z$, the average number of such maximal substring matches for a given position $i$. In principle $z$ can be large and the worst-case time complexity of our algorithm is therefore high. In practice, however, $z$ is small, independent of sequence length and substitution probability. Figure 9 shows values of $z$ for different sequence lengths and mutation frequencies.

Finally, we wanted to know how accurately our greedy heuristic approximates the exact maximum $k$-mismatch substring length. Figure 10 compares the average maximal $k$-mismatch substring length for varying substitution probabilities (a) as estimated with our heuristic and (b) calculated with a slow and exact algorithm. The figure shows that our heuristic is clearly suboptimal. But the goal of our project was not so much to precisely estimate the maximal $k$-mismatch substring lengths, but rather to define a distance measure on sequences that can be efficiently calculated and that can be used to obtain biologically meaningful results. Therefore, we think that the discrepancies between the optimal substring lengths and the values estimated by our heuristic are acceptable. Figure 10 suggests, however, that better estimates of the $k$-mismatch common substring lengths might improve the sensitivity of kmacs on divergent sequence sets since the curves for the exact solutions converge at higher substitution frequencies. In fact, on the mitochondrial genomes that we used as benchmark data, an exact algorithm led to better phylogenetic trees than our greedy heuristic (supplementary material). Therefore, it may be worthwhile to develop heuristics that approximate the maximal $k$-mismatch substring lengths more accurately.

6 CONCLUSION

Most alignment-free approaches to sequence analysis are based on exact word matches. In this paper, we presented a novel alignment-free algorithm that takes mismatches into account. This is similar in spirit to the spaced words approach that we previously proposed (Leimeister et al., 2014). But while spaced words uses word pairs of a fixed length with possible mismatches at pre-defined positions, kmacs considers maximal substring matches with $k$ mismatches at arbitrary positions. In the spaced words approach, the number of match positions in the underlying patterns is a critical parameter for the performance of the method. By contrast, in kmacs, there seems to be a fairly large range of values for $k$ that lead to high-quality results, as shown by our test results. kmacs seems therefore less sensitive to user-defined parameters.

The implementation of our approach using generalized enhanced suffix arrays enables us to analyse large sequence sets efficiently. Still, the program $K_r$ is roughly one order of magnitude faster than kmacs. One reason for this is that $K_r$ uses one single generalized suffix tree representing all input sequences, which can be calculated in time proportional to the number of sequences (Domazet-Lošo and Haubold, 2009). By contrast, kmacs calculates one generalized enhanced suffix array for each pair of sequences, so its run time is quadratic in the number of sequences. On the other hand, calculating suffix arrays for two sequences at a time is less memory consuming since one does not need to keep the suffix array for all input sequences simultaneously in main memory. Thus, our approach can be applied to larger data sets than $K_r$.

The two approaches that we developed, kmacs and spaced words, are slower than the corresponding approaches based on exact matches, ACS and the generic $k$-mer approach. Our new approaches, however, produce significantly better results than those established methods. Our test results suggest that spaced words performs slightly better than kmacs on genomic sequences, while on protein sequences, kmacs is superior.

In our program evaluation, we used DNA sequence sets with large evolutionary distances. On these sequences, our new alignment-free methods performed better than established methods that rely on exact word matches. Algorithms using exact matches, on the other hand, seem to work better on smaller evolutionary distances. $K_r$, for example, performs best on evolutionary distances of up to 0.6 substitutions per site (Haubold et al., 2009). Similarly, we observed that on closely related DNA sequences, kmacs produces sometimes best
results with \( k = 0 \), i.e. without mismatches (unpublished results). It seems therefore best to apply \textit{kmacs} to distantly related sequence sets, while methods such as \( K_r \) and ACS may be preferred on evolutionarily more closely related sequences.

In biological sequences, substitutions are more frequent than insertions and deletions. Consequently, \textit{exact} matches between local homologies can usually be extended until the first \textit{substitution} is reached. The average length of longest common substrings and of shortest unique substrings, respectively, can therefore be used to estimate \textit{substitution probabilities} (Haubold et al., 2009). This is similar for \textit{kmacs} as long as \( k \) is small enough. In this case, all \( k \) mismatches are likely to be used up in a \( k \)-mismatch common substring extension \textit{before} the first indel occurs. Thus, the average length of the longest \( k \)-mismatch common substrings depends on the frequency of mismatches and could be used to estimate substitution probabilities, just as in \( K_r \).

By contrast, if \( k \) is sufficiently large, substring matches between local homologies are essentially extended until the first \textit{indel} occurs. From this point on, the mismatch frequency is high and the remaining mismatches will be used up quickly. So in this situation, the average \( k \)-mismatch substring length depends on the frequency of \textit{indels} rather than on the frequency of substitutions. This may explain why ACS and \( K_r \) work well on closely related sequences, while \textit{kmacs} is superior on distantly related sequences where the frequency of indels may be a better measure for evolutionary distances than the frequency of mismatches.

In our study, we used alignment-free methods to reconstruct phylogenetic trees and evaluated the quality of these trees. But phylogeny reconstruction is only one important application of sequence comparison. Clustering, classification and remote-homology detection are other fundamental challenges in DNA and protein sequence analysis. With the rapidly growing size of sequence databases, alignment-free methods have become indispensable for these tasks (Hauser et al., 2013; Comin and Verzotto, 2012; Lingner and Meinicke, 2006). Given the speed of \textit{kmacs} and the quality of the phylogenetic trees that we could produce with it, our approach should be useful not only for fast phylogeny reconstruction, but also for other tasks in comparative sequence analysis.

### ACKNOWLEDGEMENT

We want to thank Bernhard Haubold, Sebastian Horwege and Manuel Landesfeind for useful comments and discussions as well as Sebastian Lindner, Martin Schönheit and Marcus Boden for providing data sets used in this study.

### REFERENCES


