Computing Phylo-*k*-mers

Nikolai Romashchenko[1⋆], Benjamin Linard[1,2⋆⋆], Fabio Pardi[1], and Eric Rivals[1⋆⋆⋆]

[1] LIRMM, MAB, UMR 5506, Univ. Montpellier, CNRS, France
nikolai.romashchenko@lirmm.fr, pardi@lirmm.fr, rivals@lirmm.fr
[2] SPYGEN, 17 Rue du Lac Saint-André, 73370 Le Bourget-du-Lac, France
Benjamin.Linard@lirmm.fr

Abstract. Phylogenetically informed k-mers, or phylo-k-mers for short, are k-mers that are predicted to appear within a given genomic region at predefined locations of a fixed phylogeny. Given a reference alignment for this genomic region and assuming a phylogenetic model of sequence evolution, we can compute a probability score for any given k-mer at any given tree node. The k-mers with sufficiently high probabilities can later be used to perform alignment-free phylogenetic classification of new sequences — a procedure recently proposed for the phylogenetic placement of metabarcoding reads and the detection of novel virus recombinants. While computing phylo-k-mers, we need to consider large numbers of k-mers at each tree node, which warrants the development of efficient enumeration algorithms.
We consider a formal definition of the problem of phylo-k-mer computation: How to efficiently find all k-mers whose probability lies above a user-defined threshold for a given tree node? We describe and analyze algorithms for this problem, relying on branch-and-bound and divide-and-conquer techniques. We exploit the redundancy of adjacent windows of the alignment and the structure of the probability matrix to save on computation. Besides computational complexity analyses, we provide an empirical evaluation of the relative performance of their implementations on real-world and simulated data. The divide-and-conquer algorithms, which to the best of our knowledge are novel, are found to be clear improvements over the branch-and-bound approach, especially when a large number of phylo-k-mers are found.

Keywords: phylo-k-mers, algorithms, enumeration, phylogenetics, metabarcoding, NGS, evolution

1 Introduction

Alignment-free approaches in bioinformatics are motivated by the fact that sequence alignment is a complex task, requiring the use of memory and time-consuming algorithms. Moreover, alignments are potentially inaccurate, sensitive

⋆ NR is supported by a fellowship from French Ministery (MNERT).
⋆⋆ BL funded by Plan de relance ANR-SPYGEN LS243173.
⋆⋆⋆ ER thanks funding from European ITN ALPACA project.

to sequencing errors, and difficult to apply to genomes with permuted structures [20]. Many alignment-free methods for solving various problems in bioinformatics (e.g., *de novo* assembly, genome comparison, read correction, read clustering) rely on the decomposition of a sequence into its constituent k-mers, that is, its substrings of length k.

Recently, a probabilistic extension of the notion of k-mers was proposed [10, 17]. In this development, many more k-mers are inferred from a set of reference sequences beyond the ones that are actually within those sequences. This inference aims at predicting k-mers that may be present in relatives of the reference sequences (e.g., within their ancestors, or within "cousin" sequences). Moreover, for any given location in the phylogeny of the reference sequences, one can estimate the probability of observing any given k-mer, meaning that probability scores can be assigned to the inferred k-mers. Key to this inference are probabilistic models of sequence evolution, which rely on a phylogenetic tree describing the evolutionary history of the reference sequences. The inferred k-mers are intended to be informative about the phylogenetic origin of newly-observed sequences containing them. For these reasons they are called *phylo-k-mers*.

Every phylo-k-mer w is associated with scores describing how probable w is to appear at a predefined set of nodes in the reference phylogeny (more detail in the Preliminaries). These scores can be used to determine the likely phylogenetic origin of any given *query* sequence, while avoiding the need to align the query to the reference sequences. This idea was recently applied to phylogenetic placement of metabarcoding reads [10] and the detection and analysis of virus recombinants composed of fragments from different viral types [17].

The main bottleneck of this technique lies in the very large number of phylo-k-mers, which comes from the fact that we need to consider up to 4^k k-mers for DNA and 20^k for protein sequences. Although we can reduce this number by only considering phylo-k-mers with probability scores above a certain threshold, practical threshold values are typically low. Thus, finding phylo-k-mers remains computationally challenging. While previous works only considered the accuracy and speed of sequence classification based on already computed phylo-k-mers [10, 17], here we focus on algorithms for computing phylo-k-mers.

In the following, we consider a number of algorithms for this problem. While one of these algorithms has already been described to some degree in the literature (e.g., [10,12,14,16]), the others are novel. We analyze the complexities of all the presented algorithms and compare their running times over simulated and real-world datasets. Both the theoretical analyses and the empirical evaluations show that the new algorithms may be significant improvements over the existing ones, especially when a large number of phylo-k-mers must be output.

Related works A problem similar to phylo-k-mer computation arises in the context of sequence motifs, precisely of Position-Specific Scoring Matrices (PSSMs), also known as Position Weight Matrices (PWMs) or weighted patterns. PSSMs represent DNA and protein sequence motifs (e.g., transcription factor binding sites) as a matrix of probabilities for each nucleotide, or amino acid, at each po-

sition in the motif. An important problem is to find significant matches of such weighted patterns in collections of genome-sized sequences. In existing algorithmic solutions to this problem, one of the preliminary steps is to enumerate all possible motif instances that reach the threshold score for a given PSSM. This step is similar to the problem of phylo-k-mer computation, with some important differences that we discuss below. Previous literature showed that the tree of all prefixes of full-length sequences with high-enough score can be explored in a depth-first [12, 16] and breadth-first [13, 14] manner.

However, in the context of phylo-k-mers, the computation is more challenging: the PSSM-based approaches only involve a single execution per profile, and the number of profiles to process is usually in the hundreds [5, 9]; on the other hand, computing phylo-k-mers may well require processing millions of matrices, as it must process each of the k-wide sub-matrices of several input matrices originating from different parts of the reference phylogeny. Another difference is that, for phylo-k-mers, score threshold values are typically much lower than for PSSM matching, meaning that a larger fraction of the possible k-mers can reach the threshold. Finally, phylo-k-mer computation assumes processing matrices related to each other, both because k-wide sub-matrices overlap, and because of the phylogenetic relatedness of the input matrices. We exploit the overlap between sub-matrices to improve running time of phylo-k-mer computation.

2 Preliminaries

2.1 Notation

Let Σ be a finite ordered alphabet of cardinality σ. We consider strings (or sequences) over alphabet Σ. Let k be a positive integer. Let Σ^k denote the set of all possible strings of length k over Σ. Given a string s, the length of s is denoted by $|s|$. For any two integers $1 \leq i \leq j \leq |s|$, s_i denotes the i^{th} letter of s, and the substring of s starting in position i and ending at position j is denoted by $s_i \dots s_j$. A substring $s_i \dots s_j$ is a prefix of s if $i = 1$, and a suffix of s if $j = |s|$. For a set X, $|X|$ denotes the number of elements in X.

We consider matrices whose rows are indexed by symbols of the alphabet Σ and whose columns are indexed as the positions of a multiple alignment. A column stores the probability of occurrences of each possible symbol (a state in phylogenetic terms) at that position. Hence, we term such matrices *probability matrices* since the values of a column sum to one. For a $\sigma \times m$ probability matrix P, $P_{\alpha,j}$ denotes the element on row α (with $\alpha \in \Sigma$) and column j of P (with $1 \leq j \leq m$); the same element is denoted by P_{ij} if α is the i-th element of Σ. For two integers i, j such that $1 \leq i \leq j \leq m$, $P[i : j]$ denotes the matrix P restricted to columns from i to j included.

2.2 Phylo-k-mers at a glance

Consider a multiple alignment of reference sequences and a phylogenetic tree $T = (V, E)$ describing the evolutionary history leading up to the reference sequences. We add to T a set of nodes V', representing sequences that are unknown

relatives of the reference sequences. (See Figure 6 in Appendix for an example.) Let m be the number of columns (sites) in the alignment. For each node $u \in V'$, we compute a $\sigma \times m$ probability matrix P^u describing the probability at u of any state in Σ, at any site in the alignment, conditional to the sequences observed at the leaves of T (i.e., the aligned reference sequences). P^u can be derived from the tree likelihood conditional to the states in Σ by applying Bayes' theorem, which is standard in phylogenetics (see, e.g., section 4.4.2.1 in [19]). Then, the complexity of computing all matrices P^u is equal to that of computing conditional tree likelihoods across all tree nodes, which for a constant-size alphabet can be done in $O(|V \cup V'| \cdot m)$ time [2] with Felsenstein's algorithm [4].

Given P^u, we can then define a probability score $S^u(w)$ associated to any given k-mer w and to the node u. See Definition 1 below for a definition of $S^u(w)$ (where the superscript is dropped for simplicity). Informally, $S^u(w)$ approximates the probability of w to appear in a sequence positioned at node u, based on the chosen model of sequence evolution and on the sequences at the leaves of T. We call the pair $(w, S^u(w))$ a phylo-k-mer.

The interest of phylo-k-mers is that finding the nodes u that maximize the product of $S^u(w)$ over all k-mers in a query sequence provides a good estimate of its evolutionary origin [10,17]. Moreover, this can be computed without aligning the query to the reference sequences, making this approach very scalable to large numbers of queries. For a detailed treatment of phylo-k-mers, see [15]. While the matrix P^u and score function S^u are relative to a particular node u, in the following we assume that the node u is fixed, and therefore omit this dependency. We simply write P and S.

2.3 The problem of phylo-k-mer computation

Here, we study the problem of enumerating k-mers and their scores relative to a probability matrix P and a threshold score value $\varepsilon \in [0, 1)$. P contains probabilities $P_{\alpha,j}$ of observing different states $\alpha \in \Sigma$ at every site j of the multiple alignment. Starting from an alignment site j, or *position* j, we can calculate the score of a k-mer $w = w_1 w_2 \dots w_k$ for this position by taking the product of corresponding probabilities: $S(w, j) = P_{w_1,j} \cdot P_{w_2,j+1} \cdot \ldots \cdot P_{w_k,j+k-1}$. We say that w *obtains the score of* $S(w, j)$ *at position* j. Since the number of possible k-mers grows exponentially with k, it is challenging to enumerate and store all k-mers for k sufficiently large. To overcome this, we only consider k-mers that obtain scores greater than ε for at least one position. For such a k-mer w, we say that w *reaches the threshold at position* j if $S(w, j) > \varepsilon$. The final score $S(w)$ is the maximum of $S(w, j)$ obtained among all positions. Definition 1 formalizes this problem.

Definition 1 (Phylo-k-mer Computation).

Input: *An integer $k > 1$; a $\sigma \times m$ probability matrix P; a threshold value $\varepsilon \in [0, 1)$.*

Output: *All pairs* $\big\{(w, S(w)) \mid w \in \Sigma^k : S(w) > \varepsilon\big\}$, *where*

$$S(w) := \max_{l=1}^{m-k+1} \Big\{ \prod_{j=1}^{k} P_{w_j, l+j-1} \Big\}.$$

3 Algorithms

Phylo-k-mer computation has been implemented in RAPPAS [10] but has not been described explicitly. Here, we describe an algorithm similar to the one of RAPPAS and present new algorithms for this problem. All described algorithms approach the problem window-by-window: given a window $W = P[j : j + k - 1]$ of k consecutive columns in P, we list all k-mers that reach the threshold for the window, as well as their scores. Let $\mathcal{Z}$ be the set of such k-mers for the window W. If $w \in \mathcal{Z}$, we call w *alive* in the window, and we call it *dead* otherwise. Then, we can obtain the solution for the global matrix P by simply taking the union of sets $\mathcal{Z}$ for every window and setting the score of each k-mer to the maximum score obtained across all windows.

In the analysis of the algorithms, we adopt the word-RAM model of computation. It assumes operating on words of size b and performing arithmetic and bitwise operations in constant time [6]. Also, we assume that the alphabet size σ is constant. Finally, we assume that any k-mer can be represented with a constant number of machine words, implying $b = \Theta(\log \sigma^k)$. Those assumptions imply that we can operate on k-mers (e.g., writing a k-mer to memory) in constant time.

3.1 Branch-and-bound

RAPPAS applied a branch-and-bound-based algorithm. Given a window W, the algorithm iterates over possible prefixes in a depth-first manner. For a prefix $p = w_1 \dots w_l$ with a score $\prod_{j=1}^{l} W_{w_j,j} > \varepsilon$, it expands p by one symbol and checks whether the score of the expanded prefix also reaches the threshold. As soon as a prefix obtains a score $\leq \varepsilon$, such a prefix is rejected. Prefixes of length k with their scores are saved as a result.

This algorithm can be naturally improved with the lookahead bound technique (introduced in [18], also used in [1, 7, 12]). Consider a lookahead bound array L of elements $L_j = \prod_{h=j+1}^{k} \max_{a \in \Sigma} W_{a,h}$ giving maximum possible scores achieved in W by suffixes of different lengths. Then, a prefix $p = w_1 \dots w_l$ of length l can be rejected if $\prod_{j=1}^{l} W_{w_j,j} \leq \varepsilon / L_l$. By analogy with k-mers, we call p *alive* if its score reaches ε / L_l, and *dead* otherwise. Note that a prefix is alive if and only if it is the prefix of an alive k-mer, i.e., an element of $\mathcal{Z}$.

Algorithm 2 in Appendix gives the pseudocode of the recursive depth-first branch-and-bound algorithm. Similar algorithms were described for preprocessing PSSMs in depth-first [12, 16] and breadth-first [13, 14] manners. In some cases (e.g., [12]), the columns of the PSSM were ordered by conservation to

facilitate early rejection of prefixes. This idea can easily be adapted for phylo-k-mer computation, by ordering the columns in each window by the entropy of the probability distribution that they define. However, in practice we did not find this to be worth the computational overhead it involves (see Figure 8 in Appendix).

Theorem 1. *Depth-first branch-and-bound runs in $\mathcal{O}(k \cdot |\mathcal{Z}|)$ time for one window of k columns.*

Theorem 1 shows the worst-case complexity of the branch-and-bound to be $\mathcal{O}(k \cdot |\mathcal{Z}|)$ (see Appendix for the proof). However, the algorithm achieves optimal best-case complexity: consider $\varepsilon = 0$ and W consisting of strictly positive probabilities, for which $|\mathcal{Z}| = \sigma^k$. The algorithm visits $\sum_{j=0}^{k} \sigma^j = (\sigma^{k+1} - 1)/(\sigma - 1) = \Theta(\sigma^k) = \Theta(|\mathcal{Z}|)$ nodes; including preprocessing time, it takes $\Theta(k + |\mathcal{Z}|) = \Theta(|\mathcal{Z}|)$ time in the best case. Finally, we note that it is possible to construct examples for which $|\mathcal{Z}| = \Theta(k^c)$ for a small constant c, and branch-and-bound runs in $\Theta(k^{c+1}) = \Theta(k \cdot |\mathcal{Z}|)$, showing that the upper bound in Theorem 1 is tight in these cases. We present one such example in Appendix.

3.2 Divide-and-conquer

We present a new algorithm for the problem of phylo-k-mer computation. It applies the divide-and-conquer technique to compute scores of prefixes and suffixes for a given window W of size k. It also relies on a score bounding technique similar to the one discussed above. Consider the array $\{\max_{a \in \Sigma} W_{a,j} : j = 1 \dots k\}$ giving maximum score values for every column. Then, let M be a data structure answering range product queries $M(j_1 : j_2)$ in constant time:

$$M(j_1 : j_2) = \prod_{l=j_1}^{j_2} \max_{a \in \Sigma} W_{a,l}$$

We start with constructing M for W, which can be done in time linear in the size of W. Then, we split W into two subwindows of sizes $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$. We compute L, defined as the list of $\lfloor k/2 \rfloor$-mers that reach the score of $\varepsilon_l = \varepsilon / M(\lfloor k/2 \rfloor + 1 : k)$ in the left subwindow. Similarly, we compute R, the list of $\lceil k/2 \rceil$-mers that reach the score of $\varepsilon_r = \varepsilon / M(1 : \lfloor k/2 \rfloor)$ in the right subwindow. Note that every $\lfloor k/2 \rfloor$-mer in L must be a prefix of at least one alive k-mer, and every $\lceil k/2 \rceil$-mer in R is a suffix of an alive k-mer. The procedure described above is applied recursively to every subwindow until, at the bottom of the recursion, we process a column j and select 1-mers reaching the score of $\varepsilon / \prod_{l=1, l \neq j}^{k} \max_{a \in \Sigma} W_{a,l}$.

We combine the results of the recursive calls as follows: if $|L| < |R|$, swap them; sort R (the smaller of the two lists) by score. Finally, for every $l \in L$, consider the elements $r \in R$ in descending order of scores; include the sequence obtained by concatenating l and r in the output, until the concatenated sequences are alive. Algorithm 1 gives the pseudocode of this algorithm.

Algorithm 1: Divide-and-conquer

Input : A $\sigma \times k$ probability matrix W, and a threshold ε
Output: $\{(w, s(w)) : s(w) > \varepsilon\}$, where $s(w)$ denotes the score of w in W.

Precompute M
return DC(1, k, ε)
/* The function below lists all the h-mers reaching the score of ε' in a window starting at site j */
Function DC(j, h, ε')**:**
  $Z \leftarrow$ empty list; $swapped = false$
  if $h = 1$ **then**
    return $\{(i-1, W_{i,j}) : W_{i,j} > \varepsilon'$ for $i \leftarrow 1 \dots \sigma\}$
  else
    $\varepsilon_l \leftarrow \varepsilon' / M(j + \lfloor h/2 \rfloor : j + h - 1)$; $\varepsilon_r \leftarrow \varepsilon' / M(j : j + \lfloor h/2 \rfloor - 1)$
    $L \leftarrow$ DC(j, $\lfloor h/2 \rfloor$, ε_l)
    $R \leftarrow$ DC($j + \lfloor h/2 \rfloor$, $\lceil h/2 \rceil$, ε_r)
    if $|L| < |R|$ **then** Swap L and R; $swapped = true$;
    Sort R by score
    foreach $(l, s_l) \in L$ **do**
      foreach $(r, s_r) \in R$ **do**
        if $s_l \cdot s_r \leq \varepsilon'$ **then break** ;
        // Concatenate l and r (in their original order):
        $x \leftarrow r \cdot 2^{\lceil \log_2 \sigma \rceil \lfloor h/2 \rfloor} + l$ **if** $swapped$ **else** $l \cdot 2^{\lceil \log_2 \sigma \rceil \lceil h/2 \rceil} + r$
        $Z.add(\{x, s_l \cdot s_r\})$
    return Z

Theorem 2. *The time complexity of Algorithm 1 is* $\mathcal{O}(k\sigma^{k/2} + |\mathcal{Z}|)$.

Theorem 2 (see Appendix for the proof) gives an upper bound for running time of Algorithm 1 as a function of the output size. Intuitively, the algorithm achieves linear complexity in output size for $|\mathcal{Z}|$ sufficiently large. This can be illustrated by the same example as for branch-and-bound: if $\varepsilon = 0$ for W of positive values, then all σ^h h-mers are alive for every recursive call. It is then easy to see that the top call runs in $\Theta(\sigma^k)$ time, while all other calls take $\Theta(k\sigma^{k/2})$ in aggregate, giving a total runtime of complexity $\Theta(|\mathcal{Z}|) = \Theta(\sigma^k)$.

3.3 Divide-and-conquer with Chained Windows

While the problem of computing phylo-k-mers (Definition 1) is defined for a $\sigma \times m$ matrix containing many $\sigma \times k$ windows, the algorithms described above only consider one window at a time. Thus, they ignore an important property of the sequence of windows of P: two adjacent windows share $(k-1)$ identical columns, meaning that some computation is redundant. Based on this observation, we suggest an improvement to the divide-and-conquer algorithm that is illustrated on Figure 1.

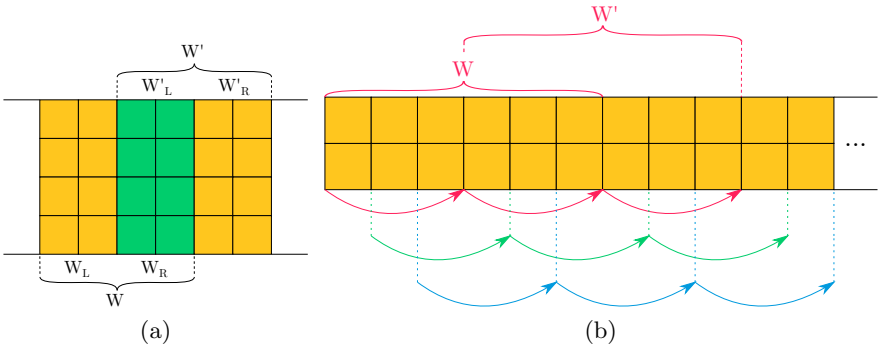


Fig. 1: Illustrations for the divide-and-conquer algorithm with Chained Windows for even k. (a) For $k = 4$ and $\sigma = 4$, two windows W and W' at a distance of $(k/2) = 2$ from each other share $(k/2) = 2$ columns. Thus, the $(k/2)$-mers alive in W_R or W'_L can be computed with a single recursive call. (b) An example of three chains (colored in red, green, and blue) of windows for $k = 6$ and $\sigma = 2$. The arrows indicate the starting positions of the different windows within the same chain. The curly braces indicate the first two windows of the red chain. In this example, all possible windows are covered with three chains.

We explain the idea for specific input and later will show how to generalize it to any input. Let k be an even value, and let the matrix P be such that $\max_a P_{a,j}$ is constant, $\forall j \in \{1, \dots, m\}$. Then, local thresholds ε_l and ε_r for a fixed recursion level are equal and constant for all windows. Consider a window W at position j, for which we recursively process its right subwindow $W[k/2+1 : k]$, obtaining the list R of alive $(k/2)$-mers and their scores $> \varepsilon_r$. Then, the list R is identical to the list L' of alive $(k/2)$-mers for the left subwindow $W'[1 : k/2]$ of another window W' starting at position $(j + k/2)$: it corresponds to the same range of columns (see 1a) and is computed for the same threshold. Naturally, we can reuse R to compute the phylo-k-mers of W'. This allows us to make only one top-level recursive call for W' instead of two (1a). We iterate over windows with a step of $k/2$, always keeping the list R of the preceding window for the next one. A sequence of windows at a distance of $k/2$ from each other is called *a chain* of windows. We need to process $k/2$ such chains starting at positions $1, 2, \dots, k/2$ to cover all windows of P. 1b illustrates this idea. Note that we still have to make both recursive calls for the first window of every chain.

The described example relies on the assumption that the threshold ε_r computed for W is equal to the threshold ε_l computed for W', which from here onwards we call ε'_l, to distinguish it from the threshold for the left subwindow of W. This allowed us to assume $R = L'$, where L' is the list of alive $(k/2)$-mers for the left subwindow of W'. Of course, ε_r and ε'_l are generally not equal, meaning that $R \neq L'$. However, it is easy to see that one of these lists is always contained in the other: if $\varepsilon'_l < \varepsilon_r$, then R is a subset of L', and vice versa otherwise. To be

sure not to lose any alive $(k/2)$-mer for the subwindow shared by W and W', we then compute the list of $(k/2)$-mers that reach $\min(\varepsilon_r, \varepsilon'_l)$. This list equals $R \cup L'$, the largest of R and L'.

The problem now becomes how to retrieve R from $R \cup L'$, when computing alive k-mers for W, and how to retrieve L' from $R \cup L'$, when computing alive k-mers for W'. This can be achieved as follows: rearrange $R \cup L'$ to separate all its elements that have a score greater than the pivot value of $\max(\varepsilon_r, \varepsilon'_l)$ (corresponding to the $(k/2)$-mers that are in the smaller of R and L') from those that have a score less or equal to $\max(\varepsilon_r, \varepsilon'_l)$ (corresponding to the $(k/2)$-mers that are only in the larger of R and L'). Once the rearrangement around the pivot is performed, retrieving R and L' from their union is trivial.

Algorithm 3 in Appendix presents the pseudocode of this algorithm for even values of k, where the PARTITION algorithm of quicksort [3] is used to rearrange $R \cup L'$, using $\max(\varepsilon_r, \varepsilon'_l)$ as pivot. Note that the algorithm substitutes the top level of the recursion, and uses the divide-and-conquer from subsection 3.2 for deeper recursive calls. The CHAIN function iterates over windows of the chain starting at position j. We assume that the data structure for range product queries is precomputed beforehand. $(k/2)$-mers for the two subwindows are combined in a way similar to the one of Algorithm 1.

Finally note that the Chained Windows technique above can also be adapted to the case of odd k, by splitting every window into three subwindows of sizes $\lfloor k/2 \rfloor$, 1, and $\lfloor k/2 \rfloor$ respectively, meaning that chains will now contain windows that are $\lceil k/2 \rceil$ sites apart from each other. We also note that the technique could in theory be adapted at every recursion level, so that only a single call to $DC(j, h, \varepsilon')$ is performed for each valid pair (j, h), with ε' set to the minimum value across all possible sub-windows from which the call to $DC(j, h, \varepsilon')$ could be executed. We leave a more thorough investigation of this idea for future work.

4 Experiments

We implemented the described algorithms (https://github.com/nromashchenko/xpas-algs as part of https://github.com/phylo42/xpas) and ran them on simulated and real-world data, using an Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz (8Mb cache size) machine with 62 Gb RAM (running under Linux 5.4.0-109-generic) and GCC 9.4.0. We measured the wall-clock time spent by every algorithm to process every window of the input matrices, and the peak memory consumption while processing all matrices.

In the first experiment, we generated a thousand random matrices of one thousand positions as follows. Every $a \in \{A, C, G, T\}$ for every position gets a random score from the uniform distribution over $[0, 1]$. Then, every column is normalized so that its values sum up to one. Note that this means that the algorithms are tested over about one million windows of size k.

In the real-world experiments, we take benchmark datasets previously used in other studies related to phylogenetic placement. Each dataset specifies a reference alignment and a reference tree. We infer two P^u matrices per branch of the

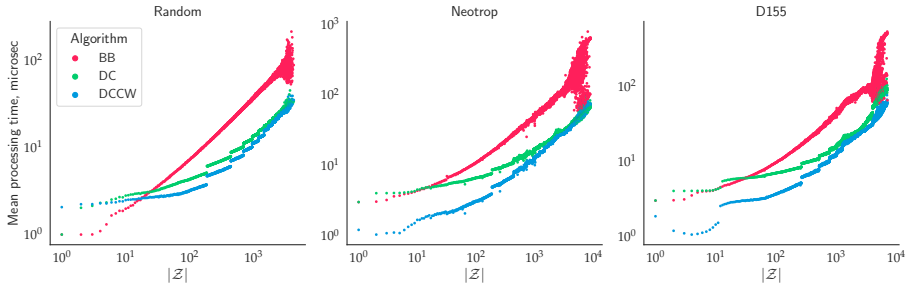


Fig. 2: Average time in microseconds to process a window of the alignment plotted against the number of phylo-k-mers alive for $k = 10$ for the three algorithms considered here: branch-and-bound (BB), divide-and-conquer (DC), and divide-and-conquer with Chained Windows (DCCW). Both axes are in log-scale.

reference tree, as it is typically done for phylogenetic placement applications [10]. The first real-world dataset, *neotrop* [11], consists of 512 Eukaryote 18S rRNA sequences of 2.8 Kbp length, resulting in 2042 matrices of size 4×2817 (≈ 5.7M k-wide sub-matrices in total). The second real-world dataset, *D155* [10], consists of 155 complete Hepatitis C Virus (HCV) genome sequences, of 9.5 Kbp length, resulting in 614 matrices of size 4×9552 (≈ 5.9M k-wide sub-matrices in total). We calculate the P^u matrices using RAxML-NG [8].

We use threshold values of $\varepsilon = (1.5/4)^k$ (the default in RAPPAS). Thus, the threshold value does not depend on the input matrix, contrary to commonly used dynamic thresholds for PSSM based on p-values. However, it depends on the length of the k-mers computed. We run algorithms for k of $6, 8, 10, 12$, which are common values for processing DNA datasets for RAPPAS (whose default value of k for DNA is 10).

Running time per window as a function of the number of alive k-mers. Figure 2 shows the mean running time per window of the three algorithms we have presented here: branch-and-bound (BB), divide-and-conquer (DC), and divide-and-conquer with Chained Windows (DCCW), plotted against the number of alive phylo-k-mers in the window, for $k = 10$. Note that many different windows may correspond to a single value of the x-axis. Each point in Figure 2 shows the average time over all windows that happened to have the same number of alive k-mers. Both axes are in log-scale. From left to right, Figure 2 shows the plot for simulated data (*Random* dataset), for *neotrop* and for *D155* datasets.

First, let us observe the relative performance of the three algorithms. In experiments both on simulated and real-world data, BB (red points) showed a better running time for k-mer-poor windows ($|\mathcal{Z}| < 25$) than DC (green points). However, BB showed a worse running time for k-mer-rich windows. Let us now compare DC (green points) against DCCW (blue points). For most values of $|\mathcal{Z}|$, DCCW showed better or similar mean running time compared to DC. For

real-world datasets, the gain in running time for DCCW is higher for k-mer-poor windows than for k-mer-rich windows. The stepwise behavior of these algorithms' running time (not happening for BB) is probably due to the allocation of additional memory needed to combine the results of the recursive calls. DCCW showed a lower running time than BB for most values of $|\mathcal{Z}|$ in all experiments.

As for the dependence of mean processing times on $|\mathcal{Z}|$, note that if we keep k constant (as done in Figure 2), the time complexity of BB is $\Theta(|\mathcal{Z}|)$ (because of Theorem 1, and because every element of $\mathcal{Z}$ is part of the output). The linear dependence of BB (red points) on $|\mathcal{Z}|$ is somewhat more visible in the *random* dataset than in the real-world datasets. As for the two divide-and-conquer algorithms, for low values of $|\mathcal{Z}|$, the runtime seems to be dominated by a term that is constant in $|\mathcal{Z}|$, which is consistent with the analysis provided in Theorem 2.

Interestingly, we remark a strong spread of the points for very high values of $|\mathcal{Z}|$ (extreme right of each panel in Figure 2), which is mostly visible for BB but also affects the other two algorithms. This is due to the fact that for very large values of $|\mathcal{Z}|$, only a few windows contribute to the computation of the mean processing time. For this reason, the computed means have an increasingly large variance. If we exclude large values of $|\mathcal{Z}|$, a large number of windows contribute to the computation of the mean processing time for most other parts of the plot. To check this, Figure 7 in Appendix plots the number of windows contributing to each value of $|\mathcal{Z}|$. The phenomenon is particularly strong for the real-world datasets, which usually only have one, two, or three windows contributing to the means for $|\mathcal{Z}| > 7500$ (Neotrop) and $|\mathcal{Z}| > 5500$ (D155).

Figure 7 also allows us to appreciate the difference between the simulated and the real-world datasets. Compared to the simulated dataset, the real-world datasets (especially D155) contain an over-representation of windows contributing with a large number of alive k-mers. Despite these differences, the three panels in Figure 2 are fairly similar. The plot for the random dataset offers a somewhat less noisy version of the other two plots.

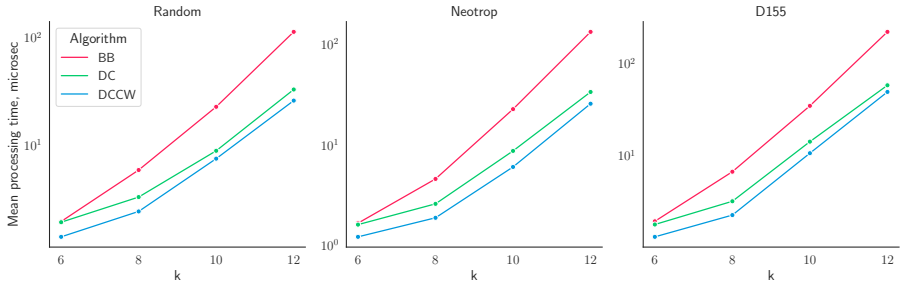


Fig. 3: Time (in microseconds, log-scale) to process a window for different values of k, averaged across all windows encountered in a single dataset.

Running time over all windows. From Figure 2, we can see that the relative performance of the algorithms is dependent on the number of alive k-mers in it. In Figure 3, we look at the overall performance of the algorithms *per dataset*, averaging processing times over all windows in a single dataset. This has the effect of naturally weighting the contribution of k-mer-rich and k-mer-poor windows according to their frequency. Figure 3 shows the mean running times for different values of k, which also allows us to examine their dependence on k.

With the possible exception of DC for $k = 6$, we note that the divide-and-conquer algorithms are faster than BB across most experiments. The speed-up of DCCW over BB varies from about 1.4x (for $k = 6$) to between 4.4x and 5.2x for $k = 12$. In all three datasets, the advantage of the two versions of divide-and-conquer for k-mer-rich regions appears to far outweigh any potential disadvantage for k-mer-poor regions. As for the dependence on k, the roughly linear plot confirms the exponential dependence of running times on k (as $|\mathcal{Z}|$ is typically an exponential function of k).

Memory consumption. Memory consumption of the three algorithms is very close in practice. We provide measurements and discuss them in Appendix (see section D, Table 1).

5 Conclusion and future work

We have described the problem of phylo-k-mer computation and algorithms for solving it. We have presented an algorithm based on the divide-and-conquer approach and a variation of it that exploits the redundancy of adjacent probability matrix windows for the input alignment. To the best of our knowledge, these two algorithms are novel, even when considering a problem similar to phylo-k-mer computation arising in the literature about motif searches. Experiments on simulated and real-world data suggest that the new algorithms perform better than the previously known branch-and-bound algorithm in terms of running time, especially when a large number of phylo-k-mers must be output.

The algorithmic results presented here, paired with an effective implementation, made it possible to improve running times of RAPPAS by up to two orders of magnitude [15]. It makes it practical for the new version of RAPPAS (manuscript in preparation) to use parameter values that were hardly feasible before, e.g., values of $k > 10$. Note that all the required preprocessing steps (construction of the references, and the computation of the P^u matrices) are independent of k, so phylo-k-mer computation from P^u is indeed the bottleneck here.

One direction for further research could exploit the phylogenetic nature of the input data: for tree nodes u, u' that are closely located in the reference tree (e.g., in terms of the length of the path separating them) the corresponding probability matrices $P^u, P^{u'}$ can also be expected to be close to each other in terms of probability values, potentially giving rise to similar sets of phylo-k-mers. Because of this, it is possible to imagine a procedure to *update* the list of phylo-k-mers, as the matrix P^u is modified.

References

1. Beckstette, M., Homann, R., Giegerich, R., Kurtz, S.: Fast index based algorithms and software for matching position specific scoring matrices. BMC Bioinformatics **7**(1) (Aug 2006). https://doi.org/10.1186/1471-2105-7-389
2. Bryant, D., Galtier, N., Poursat, M.A.: Likelihood calculation in molecular phylogenetics. In: Gascuel, O. (ed.) Mathematics of Evolution and Phylogeny. Oxford university Press (2005)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms, third edition. The MIT Press, 3rd edn. (2009)
4. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. Journal of Molecular Evolution **17**(6), 368–376 (1981)
5. Fornes, O., Castro-Mondragon, J.A., Khan, A., Van der Lee, R., Zhang, X., Richmond, P.A., Modi, B.P., Correard, S., Gheorghe, M., Baranašić, D., et al.: JASPAR 2020: update of the open-access database of transcription factor binding profiles. Nucleic Acids Research **48**(D1), D87–D92 (2020). https://doi.org/10.1093/nar/gkz1001
6. Hagerup, T.: Sorting and searching on the word RAM. In: Annual Symposium on Theoretical Aspects of Computer Science. pp. 366–398. Springer (1998). https://doi.org/10.1007/BFb0028575
7. Korhonen, J., Martinmaki, P., Pizzi, C., Rastas, P., Ukkonen, E.: MOODS: fast search for position weight matrix matches in DNA sequences. Bioinformatics **25**(23), 3181–3182 (Dec 2009). https://doi.org/10.1093/bioinformatics/btp554
8. Kozlov, A.M., Darriba, D., Flouri, T., Morel, B., Stamatakis, A.: RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference. Bioinformatics **35**(21), 4453–4455 (2019). https://doi.org/10.1093/bioinformatics/btz305
9. Kulakovskiy, I.V., Vorontsov, I.E., Yevshin, I.S., Sharipov, R.N., Fedorova, A.D., Rumynskiy, E.I., Medvedeva, Y.A., Magana-Mora, A., Bajic, V.B., Papatsenko, D.A., et al.: HOCOMOCO: towards a complete collection of transcription factor binding models for human and mouse via large-scale ChIP-Seq analysis. Nucleic Acids Research **46**(D1), D252–D259 (Nov 2018). https://doi.org/10.1093/nar/gkx1106
10. Linard, B., Swenson, K., Pardi, F.: Rapid alignment-free phylogenetic identification of metagenomic sequences. Bioinformatics **35**(18), 3303–3312 (Jan 2019). https://doi.org/10.1093/bioinformatics/btz068
11. Mahé, F., de Vargas, C., Bass, D., Czech, L., Stamatakis, A., Lara, E., Singer, D., Mayor, J., Bunge, J., Sernaker, S., et al.: Parasites dominate hyperdiverse soil protist communities in neotropical rainforests. Nature Ecology & Evolution **1**(4), 1–8 (2017). https://doi.org/10.1038/s41559-017-0091
12. Martin, D., Maillol, V., Rivals, E.: Fast and accurate genome-scale identification of DNA-binding sites. In: 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). pp. 201–205 (2018)
13. Pizzi, C., Rastas, P., Ukkonen, E.: Fast search algorithms for position specific scoring matrices. In: International Conference on Bioinformatics Research and Development. pp. 239–250. Springer (2007)
14. Pizzi, C., Rastas, P., Ukkonen, E.: Finding significant matches of Position Weight Matrices in linear time. IEEE/ACM Transactions on Computational Biology and Bioinformatics **8**(1), 69–79 (Jan 2011). https://doi.org/10.1109/TCBB.2009.35

15. Romashchenko, N.: Computing informative k-mers for phylogenetic placement. Ph.D. thesis, Université Montpellier (2021), nNT: 2021MONTS113f. tel-03629440
16. Salmela, L., Tarhio, J.: Algorithms for weighted matching. In: International Symposium on String Processing and Information Retrieval. pp. 276–286. Springer (2007)
17. Scholz, G.E., Linard, B., Romashchenko, N., Rivals, E., Pardi, F.: Rapid screening and detection of inter-type viral recombinants using phylo-k-mers. Bioinformatics **36**(22-23), 5351–5360 (2020). https://doi.org/10.1093/bioinformatics/btaa1020
18. Wu, T.D., Nevill-Manning, C.G., Brutlag, D.L.: Fast probabilistic analysis of sequence function using scoring matrices. Bioinformatics **16**(3), 233–244 (2000). https://doi.org/10.1093/bioinformatics/16.3.233
19. Yang, Z.: Computational molecular evolution. Oxford University Press Oxford (2006)
20. Zielezinski, A., Vinga, S., Almeida, J., Karlowski, W.M.: Alignment-free sequence comparison: benefits, applications, and tools. Genome Biology **18**(1), 1–17 (2017). https://doi.org/10.1186/s13059-017-1319-7

A Pseudocodes

Algorithm 2: Depth-first branch-and-bound

Input : An integer $k > 0$, a $\sigma \times k$ probability matrix W, a threshold ε
Output: The list of pairs $\{(w, s(w)) : s(w) > \varepsilon\}$, where $s(w)$ denotes the score of w in W.

$Z \leftarrow$ empty list;
$L_j \leftarrow \prod_{l=j+1}^{k} \max_{a \in \Sigma} W_{a,l}$ for all $j = 1 \dots k-1$
for $i \leftarrow 1 \dots \sigma$ **do**
    BranchAndBound$(i, 1, 0, 1)$;
return Z
/* The function below considers extending a $(j-1)$-long prefix p of score s by character a_i */
Function BranchAndBound(i, j, p, s)**:**
    $p \leftarrow 2^{\lceil \log_2 \sigma \rceil} p + i - 1$ // Update the binary representation of p
    $s \leftarrow s \cdot W_{ij}$; // Update the score of the new prefix
    if $s \leq \varepsilon / L_j$ **then** // Lookahead score bound
        return
    if $j = k$ **then**
        $Z.add(\{p, s\})$ // Report the k-mer and its score
    else
        for $i' \leftarrow 1 \dots \sigma$ **do**
            BranchAndBound$(i', j+1, p, s)$

Algorithm 3: Divide-and-conquer with Chained Windows for even k

Input : A $\sigma \times m$ probability matrix P; a threshold ε
Output: A list of pairs $\{(w, s(w)) : s(w) > \varepsilon\}$ for every k-wide window of P

for $j \leftarrow 1 \dots \lfloor k/2 \rfloor$ **do** // For every chain
  $L \leftarrow$ empty list
  for $(W_{prev}, W, W_{next}) \in$ Chain(P, j) **do** // For every window
    $\varepsilon_{LB} \leftarrow \varepsilon / M_{W_{prev}}(0 : \lfloor k/2 \rfloor)$ **if** W_{prev} **else** ε // Look behind
    $\varepsilon_{LA} \leftarrow \varepsilon / M_{W_{next}}(\lfloor k/2 \rfloor + 1 : k)$ **if** W_{next} **else** ε // Look ahead
    $Z_W, L \leftarrow$ DCCW$(L, \varepsilon_{LB}, \varepsilon_{LA})$
return all lists Z_W

Function DCCW$(L, \varepsilon_{LB}, \varepsilon_{LA})$**:**
  $Z \leftarrow$ empty list; $swapped = false$
  $\varepsilon_l = \varepsilon / M(\lfloor k/2 \rfloor + 1 : k)$; $\varepsilon_r = \varepsilon / M(0 : \lfloor k/2 \rfloor)$) // Local thresholds
  if L *is empty* **then** // If W is the first window of the chain
    $L \leftarrow$ DC$(0, \lfloor k/2 \rfloor, \varepsilon_l)$
  $R \leftarrow$ DC$(\lfloor k/2 \rfloor + 1, k - \lfloor k/2 \rfloor, \min(\varepsilon_r, \varepsilon_{LA}))$
  /* Find the number of alive prefixes by partitioning L if needed. In that case, this number is found during the partition */
  $n_l \leftarrow$ Partition(L, ε_l) **if** $\varepsilon_{LB} < \varepsilon_l$ **else** $|L|$
  $n_r \leftarrow$ Partition(R, ε_r) **if** $\varepsilon_{LA} < \varepsilon_r$ **else** $|R|$
  /* Swap L and R if needed and sort */
  if $n_l > n_r$ **then**
    Swap L and R; Swap n_l and n_r; $swapped = true$
  Sort $R[1 : n_r]$ by score // Sorts only alive elements
  foreach $(l, s_l) \in L[1 : n_l]$ **do**
    foreach $(r, s_r) \in R[1 : n_r]$ **do**
      if $s_l \cdot s_r \le \varepsilon$ **then break** ;
      $x \leftarrow r \cdot 2^{\lceil \log_2 \sigma \rceil \lfloor k/2 \rfloor} + l$ **if** *swapped* **else** $l \cdot 2^{\lceil \log_2 \sigma \rceil \lceil k/2 \rceil} + r$
      $Z.add(x, s_l \cdot s_r)$
  return $Z, (L$ **if** *swapped* **else** $R)$ // Report the result and suffixes

B Computational complexity results

B.1 Complexity of the branch-and-bound algorithm

Theorem 1 *Depth-first branch-and-bound runs in $\mathcal{O}(k \cdot |\mathcal{Z}|)$ time for one window of k columns.*

Proof. Let us consider the call tree of the algorithm where every tree node of depth j corresponds to considering a prefix of length j. We call a node alive if it corresponds to an alive prefix, and dead otherwise. Let ξ_A^j and ξ_D^j be the numbers of visited nodes of depth j that are alive and dead, respectively. Trivially, $\xi_A^k = |\mathcal{Z}|$. Note that every alive prefix of length $j-1$ is extended into at least one alive prefix of length j, implying that $\xi_A^{j-1} \leq \xi_A^j$. Therefore, $\xi_A^1 \leq \xi_A^2 \leq \cdots \leq \xi_A^{k-1} \leq \xi_A^k$, and $\sum_{j=1}^k \xi_A^j \leq k\xi_A^k = k|\mathcal{Z}|$. Now, let us count dead nodes: $\xi_D^j < \sigma\xi_A^{j-1}$, and since $\xi_A^{j-1} \leq \xi_A^j$, then $\xi_D^j < \sigma\xi_A^j$. Therefore, $\sum_{j=1}^k \xi_D^j < \sum_{j=1}^k \sigma\xi_A^j = \sigma\sum_{j=1}^k \xi_A^j \leq \sigma k|\mathcal{Z}|$. Finally, the total number of visited nodes is $\sum_{j=1}^k(\xi_A^j + \xi_D^j) < k|\mathcal{Z}| + \sigma k|\mathcal{Z}| = (\sigma+1)k|\mathcal{Z}| = \mathcal{O}(k|\mathcal{Z}|)$, assuming that σ is a constant. We visit every node in constant time by virtue of the word-RAM model assumptions. Besides that, it takes $\Theta(\sigma k)$ to precompute L. Then, the total time complexity is $\mathcal{O}(\sigma k + k|\mathcal{Z}|) = \mathcal{O}(k|\mathcal{Z}|)$.

Example 1. (A case where $|\mathcal{Z}| = \Theta(k^c)$ for a small constant c, and depth-first branch-and-bound runs in $\Theta(k^{c+1}) = \Theta(k \cdot |\mathcal{Z}|)$.)

Consider the instances of the phylo-k-mer computation problem with the following form: suppose the alphabet is binary and that all the columns of P are identical, with $P_{0,j} = p > 1/2$, and $P_{1,j} = 1 - p < 1/2$. Since we are only interested in the behavior of the algorithm on a single window, we can assume P has exactly k columns. The score of any binary sequence $w \in \{0,1\}^k$ is given by:

$$S(w) = p^{k-h(w)} \cdot (1-p)^{h(w)},$$

where $h(w)$ is the number of 1s in w (or equivalently the Hamming distance between w and 0^k). Note that $S(w)$ is strictly decreasing in $h(w)$.

Now suppose that we set $\varepsilon = S(1^{c+1}0^{k-c-1}) = p^{k-c-1}(1-p)^{c+1}$, for some constant c. (Note that since c is constant and k is not, we can assume $c \ll k$.) Then a k-mer w is alive if and only if $h(w) \leq c$, i.e., it has at most c 1s. Because of this,

$$|\mathcal{Z}| = 1 + \binom{k}{1} + \ldots + \binom{k}{c} = \Theta(1) + \Theta(k) + \ldots + \Theta(k^c) = \Theta(k^c).$$

Let us now consider the set of k-mers with $h(w) = c+1$, i.e., whose number of 1s is exactly $c+1$. There are exactly $\binom{k}{c+1} = \Theta(k^{c+1})$ such k-mers. We now prove that each of these k-mers has a different dead prefix that is visited by the algorithm: Let w be such that $h(w) = c+1$ and let p_w be the maximal alive prefix of w, ending with the character preceding the last 1 in w. Because p_w is an alive prefix, it is visited by the algorithm, as well as its dead extension p_w1 (also

a prefix of w), which however is immediately recognized as dead, as it cannot be extended in any alive k-mer. Thus each of the $\Theta(k^{c+1})$ k-mers with $h(w) = c+1$ has a dead prefix p_w1 visited by the algorithm, and moreover all the prefixes p_w1 obtained in this way are clearly different, as p_w uniquely determines w.

Because the total number of visited dead prefixes for this example is bound below by a function in $\Theta(k^{c+1})$, the running time of depth-first branch-and-bound is $\Omega(k^{c+1}) = \Omega(k \cdot |\mathcal{Z}|)$. Combining this result with the statement of Theorem 1, we obtain that on this example depth-first branch-and-bound runs in $\Theta(k^{c+1}) = \Theta(k \cdot |\mathcal{Z}|)$ time.

B.2 Complexity of the divide-and-conquer algorithm

We will approach the analysis of time complexity of Algorithm 1 as follows. First, we will analyze the complexity of the sorting performed in all recursion calls. Then, we will examine the complexity of the rest: the base case and the combination of prefixes and suffixes for all recursion calls. For the first part, lines 12—13 take $\Theta(|R| \log |R|)$ time (R might be swapped with L if $|L| < |R|$). Note that, after the potential swap, $|R| = \min\{|L|, |R|\} \leq \sigma^{\lfloor h/2 \rfloor}$. From now on, we simply write "$(h/2)$" instead of $\lfloor h/2 \rfloor$ or $\lceil h/2 \rceil$ to simplify the notation since it does not change the complexity.

Lemma 1. *The total time complexity of sorting performed by Algorithm 1 for all recursion calls is $\mathcal{O}(k \cdot \sigma^{k/2})$.*

Proof. It is easy to see that any recursion call at depth d in the recursion tree (see Figure 4) involves sorting a list of $(k/2^{d+1})$-mers. Trivially, the size of this list is at most $\sigma^{k/2^{d+1}}$. Sorting it can be done in no more than $c \cdot \sigma^{k/2^{d+1}} \log \sigma^{k/2^{d+1}} = c' \cdot k/2^{d+1} \cdot \sigma^{k/2^{d+1}}$ time (for some positive constants c, c', and assuming σ is constant). Now note that at recursion depth d there are at most 2^d recursion calls, meaning that the total runtime spent for sorting at recursion depth d is $\mathcal{O}(k \cdot \sigma^{k/2^{d+1}})$ (corresponding to the rightmost column in Figure 4).

Considering all recursion levels, the total time spent on sorting is therefore $\mathcal{O}(k \cdot S)$, where $S = \sigma^{k/2} + \sigma^{k/4} + \sigma^{k/8} + \cdots + \sigma^2 + \sigma$. Now note that

$$S \;<\; \sum_{i=0}^{k/2} \sigma^i \;=\; \frac{\sigma^{k/2+1} - 1}{\sigma - 1} \;=\; \mathcal{O}(\sigma^{k/2}),$$

which concludes the proof.

Theorem 2 *The time complexity of Algorithm 1 is $\mathcal{O}(k\sigma^{k/2} + |\mathcal{Z}|)$.*

Proof. Line 7 (the base case) takes $\Theta(\sigma) = \Theta(1)$ time. Since the complexity of sorting is given by Lemma 1, we only need to estimate the complexity of the loops at lines 14—19 to complete the analysis. Note that every element of L can give rise to at most one dead h-mer, and at least one alive h-mer, meaning

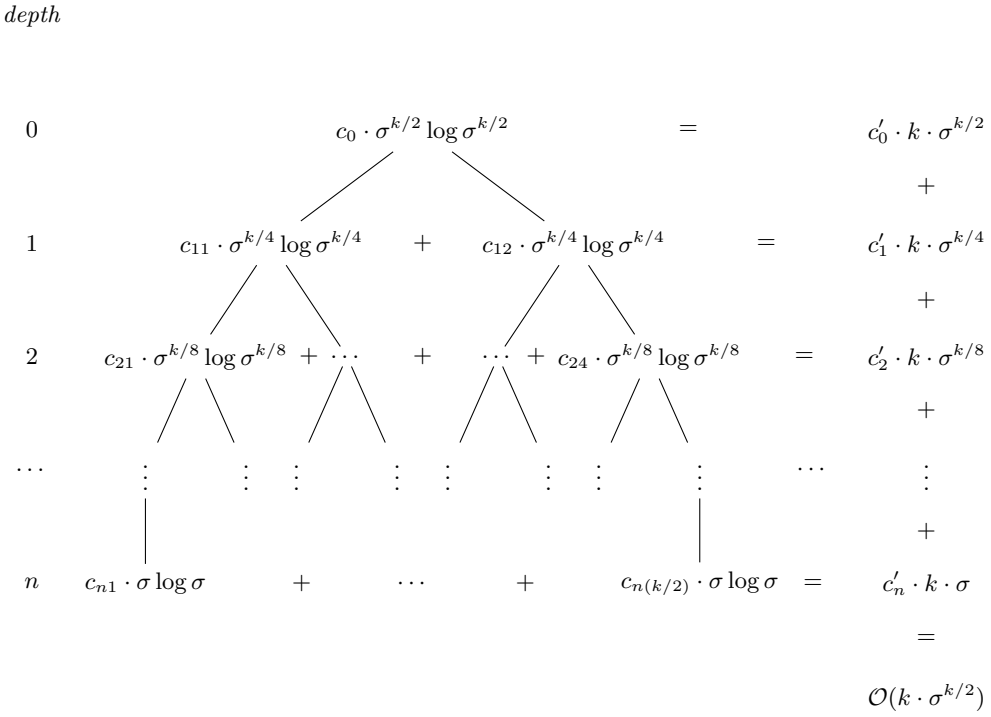


Fig. 4: Illustration for the work required to perform sorting at all recursion levels of Algorithm 1.

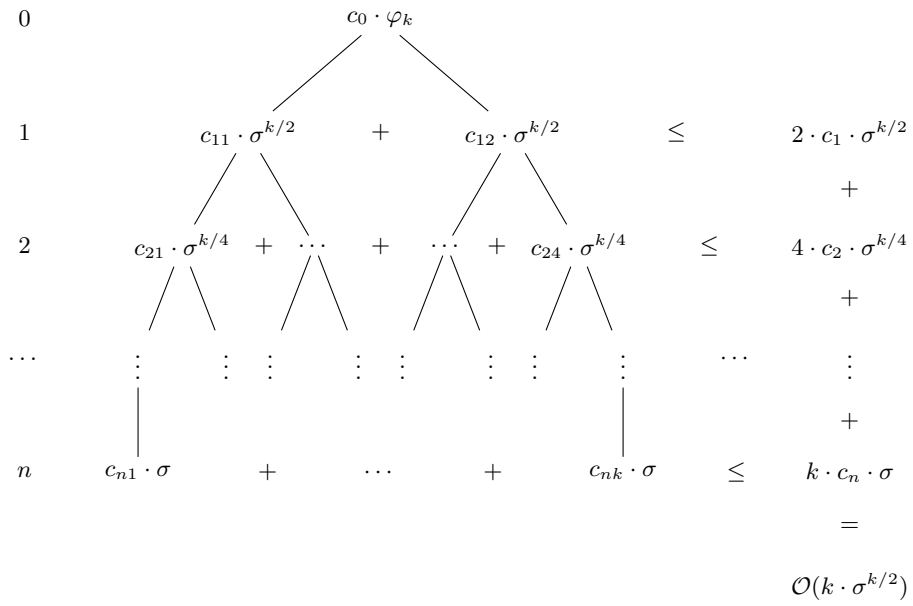


Fig. 5: Illustration of the work required to combine all alive prefix-suffix pairs for all recursive calls of Algorithm 1. If we exclude the root, the sum for the remaining nodes is $\mathcal{O}(k \cdot \sigma^{k/2})$.

that there can be at most one dead h-mer per alive h-mer. Let φ_h denote the number of alive h-mers for a recursive call acting on a window of size h, $\varphi_h \le \sigma^h$. Then, the total number of h-mers considered (dead and alive) by the loops is $\Theta(\varphi_h)$. In other words, lines 16—19 are executed $\Theta(\varphi_h)$ times, each of which takes constant time under the assumptions of the word-RAM model. Then, for the top-level recursion call, the loops take $\Theta(\varphi_k) = \Theta(|\mathcal{Z}|)$ time.

Now, let us give an upper bound for all time spent by the loops in deeper recursion calls. Each of the two recursion calls of depth 1 (when $h = k/2$) takes $\mathcal{O}(\sigma^{k/2})$ time; each of the four recursion calls of depth 2 ($h = k/4$) takes $\mathcal{O}(\sigma^{k/4})$ time, and so on (see Figure 5). In total, for all 2^i calls of depth i, the loops take $\mathcal{O}(2^i \sigma^{k/2^i})$ time, which gives us $\mathcal{O}(\sum_{i=1}^{\log k} 2^i \sigma^{k/2^i})$ for all depths (excluding the root). Let us substitute $t = \sum_{i=1}^{\log k} 2^i \sigma^{k/2^i}$. Then,

$$t = \sum_{i=1}^{\log k} 2^i \sigma^{k/2^i} \le \sum_{i=1}^{\log k} 2^i \sigma^{k/2} = \sigma^{k/2} \sum_{i=1}^{\log k} 2^i = \sigma^{k/2}\, 2\left(2^{\log k} - 1\right).$$

The last step is due to the well-known equality $\sum_{i=0}^{h-1} 2^i = 2^h - 1$. Therefore, the loops take $\mathcal{O}(t) = \mathcal{O}(k \cdot \sigma^{k/2})$ time for all recursive calls, with the exception of the root call, for which they take $\Theta(|\mathcal{Z}|)$ time. The theorem follows after Lemma 1.

C Additional figures

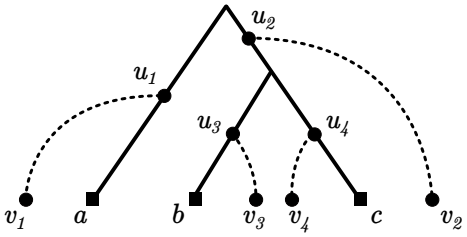


Fig. 6: A toy reference tree (solid lines) with three leaves a, b, c (filled squares), which correspond to the (observed) reference sequences, for which a multiple alignment is given as input. To this reference tree, we add the nodes in $V' = \{u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4\}$ (filled circles), representing unobserved relatives of a, b, c. Some of these nodes represent ancestral sequences (u_1, u_2, u_3, u_4), while some others represent "cousin" sequences (v_1, v_2, v_3, v_4) related to the reference tree via newly added edges (dashed lines). For each of these nodes, we can obtain probability matrices $\{P^{u_i}\}$, $\{P^{v_i}\}$, on the basis of the input alignment and of the reference tree. These matrices are the input of the phylo-k-mer computation problem.

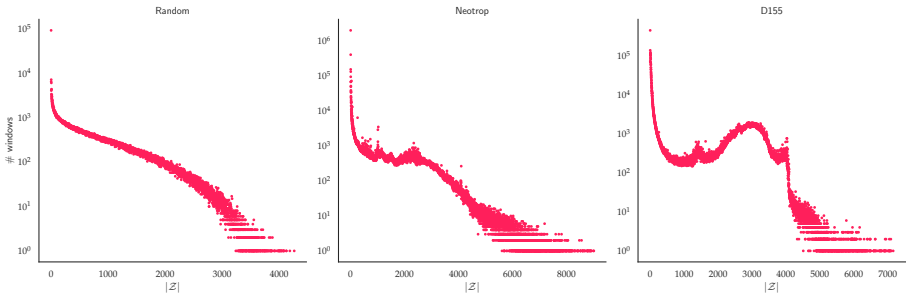


Fig. 7: Number of windows (y-axis) that have $|\mathcal{Z}|$ alive k-mers (x-axis) for the three datasets used in experiments.

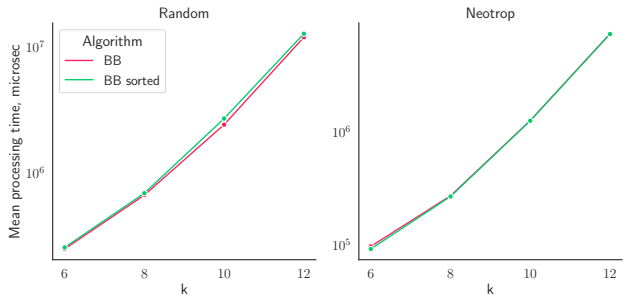


Fig. 8: Total time (in microseconds, log-scale) to process a window for different values of k by branch-and-bound on original data (BB) and on windows with sorted columns (BB sorted).

D Memory consumption

To evaluate and compare the memory requirements of the presented algorithms, we measured the peak RAM consumption as follows. For every algorithm, we ran an individual process that performed reading input data for a given dataset (or simulating input data) and phylo-k-mer computation (for $k = 10$ and the default threshold value) for all windows of all input matrices. We measured the maximal resident size reached in the process's lifetime using GNU `time`. We ran every process three times to average the measurements.

The resulting values (shown in Table 1) are virtually identical for different algorithms. While BB showed the best numbers in all experiments, the degradation of DC's and DCCW's memory consumption is under 0.01% compared to BB. This can be explained by the fact that, for all algorithms, memory consumption is dominated by the size of the input and output. For the input, we

	BB	DC	DCCW
Random	**84.00**	84.18	84.14
neotrop	**1350.60**	1350.70	1350.68
D155	**1353.73**	1353.76	1353.79

Table 1: Peak memory consumption (maximum resident set size in Megabytes) of the process performing the computation of phylo-k-mers for all input matrices of a given dataset using each of the presented algorithms. Every value is the average of measurements for three independent runs. Values in bold represent the minimal RAM consumption achieved among all algorithms for each dataset.

keep all matrices P^u in memory to optimize the overall computation for speed regardless of which algorithm is used. The output is accumulated across multiple windows of P^u, as it is required by Definition 1.