# Computing Phylo-k-mers

Nikolai Romashchenko, Benjamin Linard, Eric Rivals, Fabio Pardi

▶ **To cite this version:**

Nikolai Romashchenko, Benjamin Linard, Eric Rivals, Fabio Pardi. Computing Phylo-k-mers. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2023, 20 (5), pp.2889-2897. ⟨10.1109/TCBB.2023.3278049⟩. ⟨lirmm-03778953v2⟩

**HAL Id: lirmm-03778953**

**https://hal-lirmm.ccsd.cnrs.fr/lirmm-03778953v2**

Submitted on 15 May 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

# Computing Phylo-$k$-mers

Nikolai Romashchenko, Benjamin Linard, Eric Rivals and Fabio Pardi

**Abstract**—Finding the correct position of new sequences within an established phylogenetic tree is an increasingly relevant problem in evolutionary bioinformatics and metagenomics. Recently, alignment-free approaches for this task have been proposed. One such approach is based on the concept of phylogenetically-informative $k$-mers or phylo-$k$-mers for short. In practice, phylo-$k$-mers are inferred from a set of related reference sequences and are equipped with scores expressing the probability of their appearance in different locations within the input reference phylogeny. Computing phylo-$k$-mers, however, represents a computational bottleneck to their applicability in real-world problems such as the phylogenetic analysis of metabarcoding reads and the detection of novel recombinant viruses.

Here we consider the problem of phylo-$k$-mer computation: how can we efficiently find all $k$-mers whose probability lies above a given 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 to save on computation. Besides computational complexity analyses, we provide an empirical evaluation of the relative performance of their implementations on simulated and real-world data. The divide-and-conquer algorithms are found to surpass the branch-and-bound approach, especially when many phylo-$k$-mers are found.

**Index Terms**—phylogenetic placement, $k$-mers, enumeration algorithms, empirical runtimes, phylogeny

## 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 to sequencing errors, and difficult to apply to genomes with permuted structures [1]. 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 overlapping substrings of length $k$.

Recently, we and other authors proposed a probabilistic extension of the notion of $k$-mers [2], [3]. 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*.

• N. Romashchenko, B. Linard, E. Rivals, F. Pardi were with MAB, LIRMM, CNRS, University of Montpellier, 34095 Montpellier, France. B. Linard was also with UR MIAT, F-31320, INRAE, University of Toulouse, 31320 Castanet-Tolosan, France.
E-mails: nromashchenko@lirmm.fr, benjamin.linard@inrae.fr, rivals@lirmm.fr, pardi@lirmm.fr

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 task is also known as *phylogenetic placement*, which is an increasingly relevant problem in phylogenetic analysis and metagenomics [4]. While phylo-$k$-mers were initially developed for the phylogenetic placement of metabarcoding reads [2], they were also applied to the detection and analysis of virus recombinants composed of fragments from different viral types of the same species [3].

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, the thresholds that have been used in practical applications 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 [2], [3], 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., [2], [5], [6], [7]), 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 theoretical analyses and empirical evaluations show that the new algorithms can be significant improvements over the existing one, 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 position 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 a 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 motifs with high-enough scores could be efficiently explored in a depth-first [7], [5] and breadth-first manner [8], [6].

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 [9], [10]; 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 the running time of phylo-$k$-mer computation.

## 2 Preliminaries

### 2.1 Notation

Let $\Sigma$ be a finite ordered alphabet of cardinality $\sigma$. We consider strings (or sequences) over alphabet $\Sigma$. Let $k$ be a positive integer. Let $\Sigma^k$ denote the set of all possible strings of length $k$ over $\Sigma$. Given a string $s$, the length of $s$ is denoted by $|s|$. For any two integers $1 \le i \le j \le |s|$, $s_i$ denotes the $i^{th}$ character 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 characters of $\Sigma$ and whose columns are indexed as the positions of a multiple alignment. A column stores the probability of occurrences of each possible character 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 $\alpha$ (with $\alpha \in \Sigma$) and column $j$ of $P$ (with $1 \le j \le m$); the same element is denoted by $P_{ij}$ if $\alpha$ is the $i$-th element of $\Sigma$. For two integers $i, j$ such that $1 \le i \le j \le m$, $P[i : j]$ denotes the matrix $P$ restricted to columns from $i$ to $j$ included.

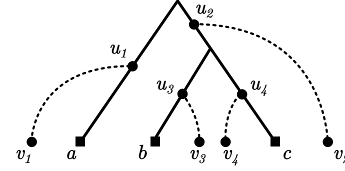


Fig. 1: A toy reference tree (solid lines) with three leaves $a, b, c$ (filled squares), which correspond to the (observed) reference sequences, for which a multiple alignment is given as input. To this reference tree, we add the nodes in $V' = \{u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4\}$ (filled circles), 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.

### 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 1 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 character in $\Sigma$, 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 character at $u$ by applying Bayes' theorem, which is standard in phylogenetics (see, e.g., section 4.4.2.1 in [11]). 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 [12] with Felsenstein's algorithm [13].

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 the evolutionary origin of the query [2], [3]. 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 [14]. While the matrix $P^u$ and score function $S^u$ are relative to a particular node $u$, in the following, we assume that 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 characters $\alpha \in \Sigma$ 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: $S(w, j) = P_{w_1,j} \cdot P_{w_2,j+1} \cdot \dots \cdot P_{w_k,j+k-1}$. We say that $w$ *obtains the score of* $S(w, j)$ *at position* $j$. We only consider $k$-mers that obtain scores greater than $\varepsilon$ 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* $\{(w, S(w)) \mid w \in \Sigma^k : S(w) > \varepsilon\}$*, 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 [2] but has not been described explicitly. Here, we describe an algorithm similar (but not equivalent) to the one of RAPPAS and present new algorithms for this problem. Unless otherwise stated, 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 $P$ 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 [15]. Also, we assume that the alphabet size $\sigma$ 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 binary representations of $k$-mers (e.g., writing a $k$-mer to memory) in constant time.

### 3.1 Branch-and-bound

Given a window $W$, a naïve algorithm, implemented in RAPPAS, 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 character and checks whether the score of the expanded prefix also reaches the threshold. As soon as a prefix obtains a score $\leq \varepsilon$, it is rejected.

This algorithm can be naturally improved with the lookahead bound technique (introduced in [16], also used in [17], [18], [5]). 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 $\varepsilon / L_l$, 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 $\mathcal{Z}$.

Fig. 2: Depth-first branch-and-bound algorithm

```
1:  procedure BranchAndBound(i, j, p, s)   ▷ Considers extending prefix p of score s by the i-th character of the alphabet
2:      p ← 2^⌈log₂ σ⌉ p + i − 1           ▷ Update the binary representation of p and its score
3:      s ← s · W_ij;
4:      if s ≤ ε/L_j then                  ▷ Lookahead score bound
5:          return
6:      end if
7:      if j = k then
8:          Add (p, s) to Z
9:      else
10:         for all i′ ← 1 . . . σ do
11:             BranchAndBound(i′, j + 1, p, s)
12:         end for
13:     end if
14: end procedure
15:
16: Z ← empty list;
17: L_j ← ∏_{l=j+1}^{k} max_{a∈Σ} W_{a,l}  for all j = 1 . . . k − 1
18: for all i ← 1 . . . σ do
19:     BranchAndBound(i, 1, 0, 1);
20: end for
21: return Z
```

Figure 2 gives the pseudocode of the recursive depth-first branch-and-bound algorithm. Similar algorithms were described for preprocessing PSSMs in depth-first [7], [5] and breadth-first [8], [6] manners. In some cases (e.g., [5]), 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 section B in Appendix).

**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 $\xi_A^j$ and $\xi_D^j$ 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, $\xi_A^1 \leq \xi_A^2 \leq \dots \leq \xi_A^{k-1} \leq \xi_A^k$, 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 $\sigma$ is a constant. We visit every node in constant time by virtue of the word-RAM model. Besides that, it takes $\Theta(\sigma k)$ to precompute $L$. Then, the total time complexity is $\mathcal{O}(\sigma k + k|\mathcal{Z}|) = \mathcal{O}(k|\mathcal{Z}|)$. □

Theorem 1 shows the worst-case complexity of the branch-and-bound algorithm to be $\mathcal{O}(k \cdot |\mathcal{Z}|)$. 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 $\Theta(k^{c+1}) = \Theta(k \cdot |\mathcal{Z}|)$ time, showing that the upper bound in Theorem 1 is tight in these cases. See Appendix A for one such example.

### 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 (see Appendix E):

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

We start with constructing $M$ for $W$, which can be done in time linear in the size of $W$. Trivially, if $M(1 : k) \leq \varepsilon$, we return an empty list as no $k$-mer is alive in the window. Otherwise, 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, while the concatenated sequences are alive. Figure 3 gives the pseudocode of this algorithm.

**Theorem 2.** *Divide-and-conquer runs in $\mathcal{O}(k\sigma^{k/2} + |\mathcal{Z}|)$ time for one window of $k$ columns.*

*Proof.* First note that both $\sigma^{\lfloor h/2 \rfloor}$ and $\sigma^{\lceil h/2 \rceil}$ are $\Theta(\sigma^{h/2})$, so we drop all notation expressing integer rounding.

Fig. 3: Divide-and-conquer algorithm

```
1:  procedure DC(j, h, ε′)    ▷ Lists all the h-mers reaching the score of ε′ in a window starting at site j
2:      Z ← empty list; swapped ← false
3:      if h = 1 then
4:          return {(i − 1, W_{i,j}) : W_{i,j} > ε′ for i ← 1...σ }
5:      end if
6:      ε_l ← ε′/ M(j + ⌊h/2⌋ : j + h − 1)
7:      ε_r ← ε′/ M(j : j + ⌊h/2⌋ − 1)
8:      L ← DC(j, ⌊h/2⌋, ε_l)
9:      R ← DC(j + ⌊h/2⌋, ⌈h/2⌉, ε_r)
10:     if |L| < |R| then
11:         Swap L and R; swapped ← true
12:     end if
13:     Sort R by score
14:     for all (l, s_l) ∈ L do
15:         for all (r, s_r) ∈ R do
16:             if s_l · s_r ≤ ε′ then break; end if
                ▷ Concatenate l and r (in their original order):
17:             if swapped then
18:                 x ← r · 2^(⌈log₂ σ⌉⌊h/2⌋) + l
19:             else
20:                 x ← l · 2^(⌈log₂ σ⌉⌈h/2⌉) + r
21:             end if
22:             Add (x, s_l · s_r) to Z
23:         end for
24:     end for
25:     return Z
26: end procedure
27:
28: Precompute M
29: return DC(1, k, ε) if M(1 : k) > ε else empty list
```

We consider a generic recursion call acting on a subwindow of size $h \leq k$ and analyse its runtime complexity. The total runtime will be obtained by summing runtimes over all recursion calls. Lines 2-12, except the calls to DC(), take constant time. Line 13 involves sorting a list of $(h/2)$-mers, whose size is trivially at most $\sigma^{h/2}$. This can be done in no more than $c\cdot\sigma^{h/2} \log \sigma^{h/2} = c' \cdot h/2 \cdot \sigma^{h/2} = \mathcal{O}(h \cdot \sigma^{h/2})$ time (for some positive constants $c, c'$, and assuming $\sigma$ is constant). We then move on to the complexity of the loops at lines 14—24 to complete the analysis of a single recursion call.

Let $\varphi_h$ denote the number of alive $h$-mers for a recursive call acting on a window of size $h$, $\varphi_h \leq \sigma^h$. Note that every $(h/2)$-mer in $L$ gives rise to at least one alive $h$-mer, and leads to considering at most one dead $h$-mer. Thus, at most one dead $h$-mer is considered per alive $h$-mer. Then, the total number of $h$-mers considered (dead and alive) by the loops is $\Theta(\varphi_h)$. In other words, lines 16—22 are executed $\Theta(\varphi_h)$ times, each of which takes constant time under the assumptions of the word-RAM model. In conclusion, overall, a single recursion call takes $\mathcal{O}(\varphi_h + h \cdot \sigma^{h/2})$ time.

The top-level recursion call takes $\mathcal{O}(|\mathcal{Z}| + k \cdot \sigma^{k/2})$ time, which is the claimed asymptotic complexity. We now show that the total time spent in deeper recursion calls is dominated by this complexity. Trivially, the runtime for a single recursion call of depth $d \geq 1$ is bounded above by

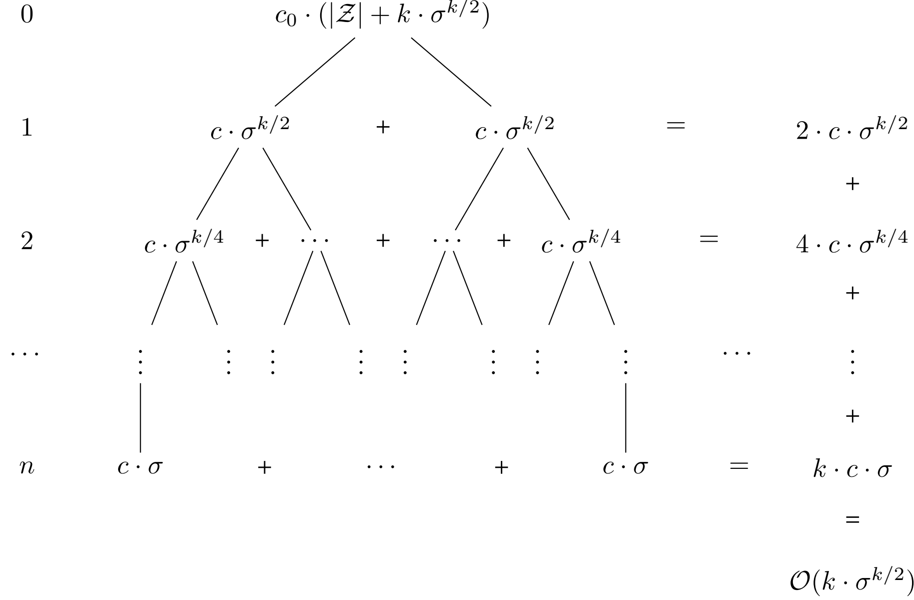


Fig. 4: Illustration of the work required by each recursion call of the algorithm in Figure 3. If we exclude the root, the sum for the remaining nodes is $\mathcal{O}(k \cdot \sigma^{k/2})$.

$c \cdot \sigma^h = c \cdot \sigma^{k/2^d}$, for some positive constant $c$ (because $\varphi_h \leq \sigma^h$ and $h \cdot \sigma^{h/2} \leq c' \cdot \sigma^h$, for $c' \geq 2$). Now note that there are at most $2^d$ calls of depth $d$, which gives us a total of $\mathcal{O}(\sum_{d=1}^{\log k} 2^d \sigma^{k/2^d})$ for all depths (excluding the top-level call; see the rightmost column in Figure 4). Because $2^d \leq k$, we can write this as $\mathcal{O}(k \cdot S)$, where $S = \sigma^{k/2} + \sigma^{k/4} + \sigma^{k/8} + \cdots + \sigma^2 + \sigma$. To conclude the proof, note that

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

□

Theorem 2 gives an upper bound for running time of the algorithm 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 $\sigma^h$ $h$-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, illustrated in Figure 5.

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, $\forall j \in \{1, \ldots, m\}$. Then, local thresholds $\varepsilon_l$ and $\varepsilon_r$ 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 Figure 5a) 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 (Figure 5a). 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 $1, 2, \ldots, k/2$ to cover all windows of $P$. Figure 5b 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 $\varepsilon_r$ computed for $W$ is equal to the threshold $\varepsilon_l$ computed for $W^+$, which from here onwards we call $\varepsilon_l^+$, 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, $\varepsilon_r$ and $\varepsilon_l^+$ 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

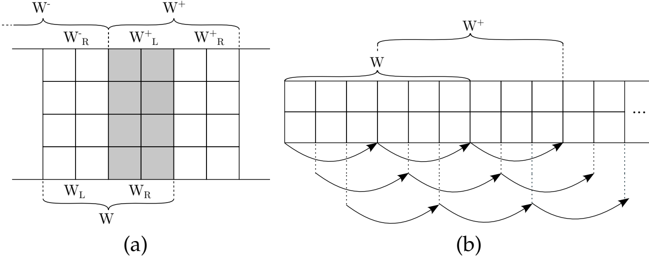


Fig. 5: Illustrations for the divide-and-conquer algorithm with Chained Windows for even $k$. (a) For $k = 4$ and $\sigma = 4$, two windows $W$ and $W^+$ at a distance of $(k/2) = 2$ from each other share $(k/2) = 2$ columns (gray area). Thus, the $(k/2)$-mers alive in $W_R$ or $W_L^+$ can be computed with a single recursive call. (b) An example of three chains 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 first chain. In this example, all possible windows are covered with three chains.

$R \cup L^+$, 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.

Figure 6 presents the pseudocode of this algorithm for even values of $k$, where the PARTITION algorithm of quick-sort [19] 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 described in subsection 3.2.

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 $\varepsilon'$ 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.

Fig. 6: Divide-and-conquer with Chained Windows for even $k$. This pseudocode is executed on the entire probability matrix $P$, and not on a window-by-window basis. See Definition 1 for the meaning of global parameters $P$ and $\varepsilon$.

```
1:  for all j ← 1 ... k/2 do
2:      L ← empty list
            ▷ for the previous, current, and next window of the chain
3:      for all (W⁻, W, W⁺) ∈ CHAIN(P, j) do
            ▷ "look behind" and "look ahead" score thresholds
4:          ε_r⁻ ← ε/M_{W⁻}(1 : k/2) if W⁻ ≠ nil else ε
5:          ε_l⁺ ← ε/M_{W⁺}(k/2 + 1 : k) if W⁺ ≠ nil else ε
6:          Z_W, L ← DCCW(L, ε_r⁻, ε_l⁺)
7:      end for
8:  end for
9:  return all lists Z_W
10:
11: procedure DCCW(L, ε_r⁻, ε_l⁺)
12:     Z ← empty list; swapped ← false
13:     ε_l = ε/M(k/2 + 1 : k)                ▷ Local thresholds
14:     ε_r = ε/M(1 : k/2)
15:
16:     if L is empty then          ▷ If W is the first window
17:         L ← DC(1, k/2, ε_l)
18:     end if
19:     R ← DC(k/2 + 1, k/2, min(ε_r, ε_l⁺))
20:     n_l ← PARTITION(L, ε_l) if ε_r⁻ < ε_l else |L| ▷ Find the number of alive prefixes by partitioning L if needed
21:     n_r ← PARTITION(R, ε_r) if ε_l⁺ < ε_r else |R|
22:
23:     if n_l > n_r then
24:         Swap L and R; Swap n_l and n_r; swapped = true
25:     end if
26:     Sort R[1 : n_r] by score
27:
28:     for all (l, s_l) ∈ L[1 : n_l] do
29:         for all (r, s_r) ∈ R[1 : n_r] do
30:             if s_l · s_r ≤ ε then
31:                 break
32:             end if
33:             x ← r · 2^{⌈log₂ σ⌉⌊k/2⌋} + l if swapped else l · 2^{⌈log₂ σ⌉⌈k/2⌉} + r
34:             Add (x, s_l · s_r) to Z
35:         end for
36:     end for
37:     return Z, (L if swapped else R)
38: end procedure
```

## 4 EXPERIMENTS

We implemented the described algorithms (https://github.com/nromashchenko/pk-algs) as part of the new software IPK (https://github.com/phylo42/IPK) 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 each, 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 edge of the reference tree, as it is typically done for phylogenetic placement applications [2] (see also Figure 1).

The first real-world dataset, *neotrop* [20], consists of 512 eukaryotic 18S rRNA sequences of 2.8 Kbp length, resulting in $2042$ matrices of size $4 \times 2817$ ($\approx$ 5.7M $k$-wide sub-matrices in total). The second real-world dataset, *D155* [2], consists of $155$ complete Hepatitis C Virus (HCV) genome sequences, of 9.5 Kbp length, resulting in $614$ matrices of size $4 \times 9552$ ($\approx$ 5.9M $k$-wide sub-matrices in total). We calculate the $P^u$ matrices using RAxML-NG [21].

We focus on threshold values of $\varepsilon = (1.5/4)^k$ as this is the default in RAPPAS as well as in IPK. 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 $k$ equal to $6, 8, 10$, and $12$, which are common values for processing DNA datasets for RAPPAS (whose default value of $k$ for DNA is 8).

### 4.1 Running time per window as a function of the number of alive $k$-mers

Figure 7 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 $k = 10$ and $\varepsilon = (1.5/4)^k$ (see Appendix F for other threshold values). Note that many different windows may correspond to a single value of the x-axis. Each point in Figure 7 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 7 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 phylo-$k$-mer-poor windows ($|\mathcal{Z}| < 25$) than DC (green points). However, BB showed the worst running time for most values of $|\mathcal{Z}|$. Let us now compare DC (green points) against DCCW (blue points). For nearly all 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 phylo-$k$-mer-poor windows than for phylo-$k$-mer-rich windows.

As for the dependence of mean processing times on $|\mathcal{Z}|$, note that if we keep $k$ constant (as done in Figure 7), 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 7), which mostly affects BB. 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 10 in Appendix C plots the number of windows contributing to each value of $|\mathcal{Z}|$. The fact that this phenomenon appears to affect BB more than the other two algorithms suggests a higher variance of its running time.

Figure 10 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 7 are fairly similar. Unsurprisingly, the plot for the random dataset offers a somewhat less noisy version of the other two plots.

### 4.2 Running time over all windows

From Figure 7, we can see that the relative performance of the algorithms is dependent on the number of alive $k$-mers in it. In Figure 8, 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 phylo-$k$-mer-rich and -poor windows according to their frequency. Figure 8 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 $1.4$x (for $k = 6$) to between $4.4$x and $5.2$x for $k = 12$. In all three datasets, the advantage of the two versions of divide-and-conquer for phylo-$k$-mer-rich regions appears to outweigh by far any potential disadvantage for phylo-$k$-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 $k$).

### 4.3 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).

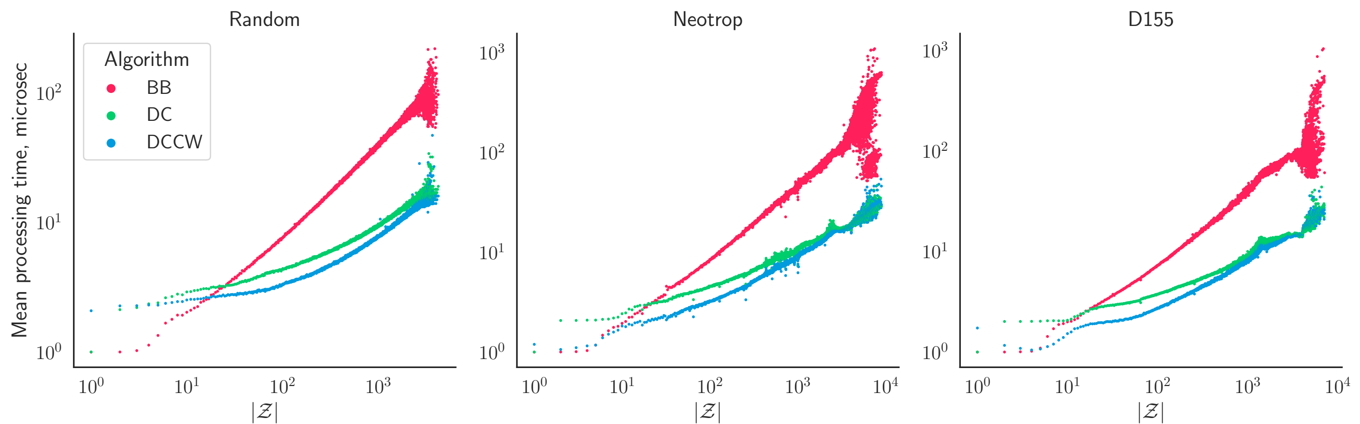


Fig. 7: Average time in microseconds to process a window of the alignment plotted against the number of phylo-$k$-mers alive for $k = 10$ and $\varepsilon = (1.5/4)^k$, for the three algorithms considered here: branch-and-bound (BB), divide-and-conquer (DC), and divide-and-conquer with Chained Windows (DCCW). Both axes are in log-scale.

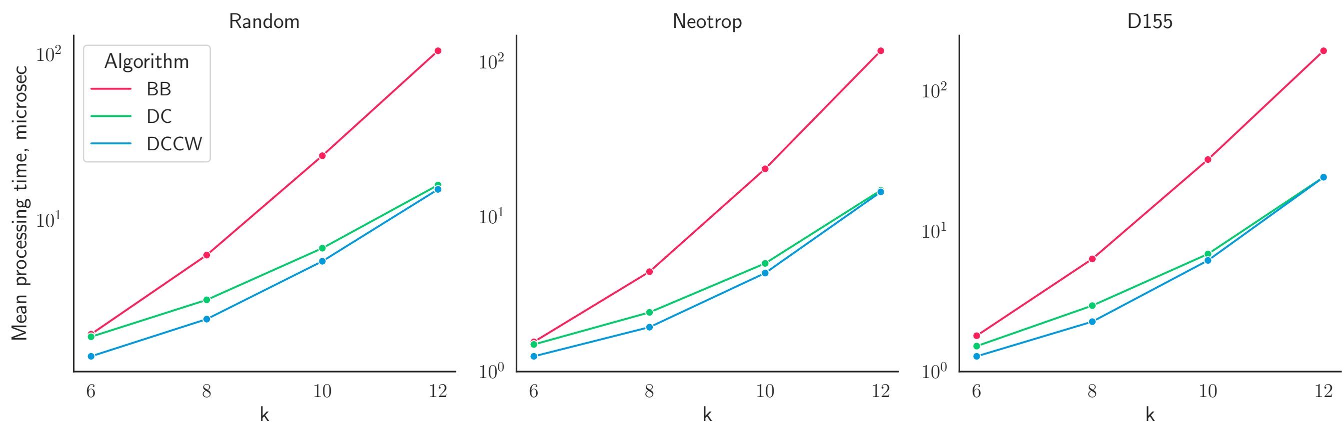


Fig. 8: Time (in microseconds, log-scale) to process a window for different values of $k$, averaged across all windows encountered in a single dataset.

## 5 Conclusion and Future Work

We have described the problem of phylo-$k$-mer computation and algorithms for solving it. In particular, we designed two algorithms based on the divide-and-conquer approach, one of which exploits the redundancy of adjacent probability matrix windows of 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 C++ implementation, made it possible to improve the running times of RAPPAS (written in Java) by two orders of magnitude [14]. It makes it practical for the successor 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 future research is to exploit the relationships between nodes encoded in the phylogeny. Let $u, v$ be tree nodes that are closely located in the reference tree (e.g., in terms of the length of the path separating them). Then, the corresponding probability matrices $P^u, P^v$ 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. The question is thus to find an algorithm that, instead of computing from scratch the phylo-$k$-mers for node $v$, updates the list of phylo-$k$-mers computed for node $u$ by taking advantage of the proximity of $P^u$ and $P^v$.

## Acknowledgments

We thank Bastien Cazaux for his comments and the valuable discussion of the algorithms and the French Ministry of Research (MNERT) for a fellowship to NR. ER is funded by the ALPACA project, which has received funding from the European Union's Horizon 2020 research and innovation

program under the Marie Skłodowska-Curie grant agreement № 956229. FP is funded by the ANR via the project Fish-Predict.

## REFERENCES


[1] A. Zielezinski, S. Vinga, J. Almeida, and W. M. Karlowski, "Alignment-free sequence comparison: benefits, applications, and tools," *Genome Biology*, vol. 18, no. 1, pp. 1–17, 2017.

[2] B. Linard, K. Swenson, and F. Pardi, "Rapid alignment-free phylogenetic identification of metagenomic sequences," *Bioinformatics*, vol. 35, no. 18, p. 3303–3312, Jan 2019.

[3] G. E. Scholz, B. Linard, N. Romashchenko, E. Rivals, and F. Pardi, "Rapid screening and detection of inter-type viral recombinants using phylo-k-mers," *Bioinformatics*, vol. 36, no. 22-23, pp. 5351–5360, 2020.

[4] L. Czech, A. Stamatakis, M. Dunthorn, and P. Barbera, "Metagenomic analysis using phylogenetic placement — a review of the first decade," *arXiv preprint arXiv:2202.03534*, 2022.

[5] D. Martin, V. Maillol, and E. Rivals, "Fast and accurate genome-scale identification of DNA-binding sites," in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2018, pp. 201–205.

[6] C. Pizzi, P. Rastas, and E. Ukkonen, "Finding significant matches of Position Weight Matrices in linear time," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 1, pp. 69–79, Jan. 2011.

[7] L. Salmela and J. Tarhio, "Algorithms for weighted matching," in *International Symposium on String Processing and Information Retrieval*. Springer, 2007, pp. 276–286.

[8] C. Pizzi, P. Rastas, and E. Ukkonen, "Fast search algorithms for position specific scoring matrices," in *International Conference on Bioinformatics Research and Development*. Springer, 2007, pp. 239–250.

[9] I. V. Kulakovskiy, I. E. Vorontsov, I. S. Yevshin, R. N. Sharipov, A. D. Fedorova, E. I. Rumynskiy, Y. A. Medvedeva, A. Magana-Mora, V. B. Bajic, D. A. Papatsenko, and 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*, vol. 46, no. D1, p. D252–D259, Nov 2018.

[10] O. Fornes, J. A. Castro-Mondragon, A. Khan, R. Van der Lee, X. Zhang, P. A. Richmond, B. P. Modi, S. Correard, M. Gheorghe, D. Baranašić *et al.*, "JASPAR 2020: update of the open-access database of transcription factor binding profiles," *Nucleic Acids Research*, vol. 48, no. D1, pp. D87–D92, 2020.

[11] Z. Yang, *Computational molecular evolution*. Oxford University Press Oxford, 2006.

[12] D. Bryant, N. Galtier, and M.-A. Poursat, "Likelihood calculation in molecular phylogenetics." in *Mathematics of Evolution and Phylogeny*, O. Gascuel, Ed. Oxford university Press, 2005.

[13] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *Journal of Molecular Evolution*, vol. 17, no. 6, pp. 368–376, 1981.

[14] N. Romashchenko, "Computing informative k-mers for phylogenetic placement," Ph.D. dissertation, Université Montpellier, 2021, https://theses.hal.science/tel-03629440.

[15] T. Hagerup, "Sorting and searching on the word RAM," in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1998, pp. 366–398.

[16] T. D. Wu, C. G. Nevill-Manning, and D. L. Brutlag, "Fast probabilistic analysis of sequence function using scoring matrices," *Bioinformatics*, vol. 16, no. 3, pp. 233–244, 2000.

[17] M. Beckstette, R. Homann, R. Giegerich, and S. Kurtz, "Fast index based algorithms and software for matching position specific scoring matrices," *BMC Bioinformatics*, vol. 7, no. 1, Aug 2006.

[18] J. Korhonen, P. Martinmaki, C. Pizzi, P. Rastas, and E. Ukkonen, "MOODS: fast search for position weight matrix matches in DNA sequences," *Bioinformatics*, vol. 25, no. 23, pp. 3181–3182, Dec. 2009.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, third edition*, 3rd ed. The MIT Press, 2009.

[20] F. Mahé, C. de Vargas, D. Bass, L. Czech, A. Stamatakis, E. Lara, D. Singer, J. Mayor, J. Bunge, S. Sernaker *et al.*, "Parasites dominate hyperdiverse soil protist communities in neotropical rainforests," *Nature Ecology & Evolution*, vol. 1, no. 4, pp. 1–8, 2017.

[21] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis, "RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference," *Bioinformatics*, vol. 35, no. 21, pp. 4453–4455, 2019.


**Nikolai Romashchenko** is a research engineer at the CNRS (French National Centre for Scientific Research) and the LIRMM (Laboratory of Computer Science, Robotics and Microelectronics of Montpellier). He received a doctoral degree in Computer Science at the University of Montpellier in 2021. His scientific interests are algorithms for bioinformatics, metagenomics, and phylogenetics, with a particular focus on alignment-free methods.

**Benjamin Linard** was a research officer at SPYGEN, a company specialising in aquatic and terrestrial biodiversity monitoring using environmental DNA, until November 2022. He is currently taking on a new position as *chargé de recherche* at the INRAE (French National Research Institute for Agriculture, Food and Environment). He received his PhD in Bioinformatics at the University of Strasbourg in 2012. He previously worked on metagenomic analysis in the Natural History Museum of London (2013-2015) and developed the idea of phylo-k-mers during his postdoc in the Computer Science Dpt of the LIRMM (2016-2019), a joint CNRS - Univ. Montpellier unit. His research focuses on the development of algorithms and software for pangenomics, comparative genomics and environmental DNA analysis.

**Eric Rivals.** After a PhD in computer science (University of Lille), Eric Rivals worked 3 years at the German Cancer Res. Center in Heidelberg in Martin Vingron bioinformatic lab. Currently, research director in the Computer Science Dpt of the LIRMM, a joint CNRS - Univ. Montpellier unit, he focuses on algorithms for bioinformatics and their applications in genomics, transcriptomics, and cancer research. Besides algorithm design, he also developed software for bioinformatic analyses of large deep sequencing data. For instance, LoRDEC, one of the first efficient long read error correction program, is widely used in genome and transcriptome sequencing projects around the world (Salmela Rivals, 2014). He recently co-published and co-patented a diagnostic method for a brain cancer based on the detection of epitranscriptomic modifications on RNA (Relier et al. 2022). ER served as Head of the Computer Science Dpt of the LIRMM (2007-2010), Head of the national network for molecular bioinformatics of the CNRS from (2010-2015), and Head of the Computational Biology Institute in Montpellier (2015-2019). Since 2016, he is also the scientific leader of the ATGC bioinformatic service platform, which attracts more than 10,000 unique users every year.

**Fabio Pardi** is a *chargé de recherche* at the CNRS (French National Centre for Scientific Research) and the LIRMM (Laboratory of Computer Science, Robotics and Microelectronics of Montpellier). He received his PhD in Bioinformatics at the University of Cambridge and the European Bioinformatics Institute in 2009. His research interests include mathematical models and computationally-efficient methods for the study of evolution at every time scale.

# Appendix A
## Worst-case performance of depth-first branch-and-bound

Here, we show 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,

$$\begin{aligned} |\mathcal{Z}| &= 1 + \binom{k}{1} + \ldots + \binom{k}{c} \\ &= \Theta(1) + \Theta(k) + \ldots + \Theta(k^c) = \Theta(k^c). \end{aligned} \quad (1)$$

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.

# Appendix B
## Branch-and-bound with sorted columns

We compared the performance of the branch-and-bound algorithm as described in the main text against its modified version. Namely, we sorted columns of the matrix by their entropy before the phylo-$k$-mer computation. Thus, branch-and-bound visits the columns not in their original order but in descending order of their informativeness. To reduce the computational overhead, we did not rearrange the columns in memory but kept their original order while giving the algorithm the appropriate order to visit them. The latter is to avoid rearranging the same $(k-1)$ columns in memory when switching from one $k$-sized window to the next one.

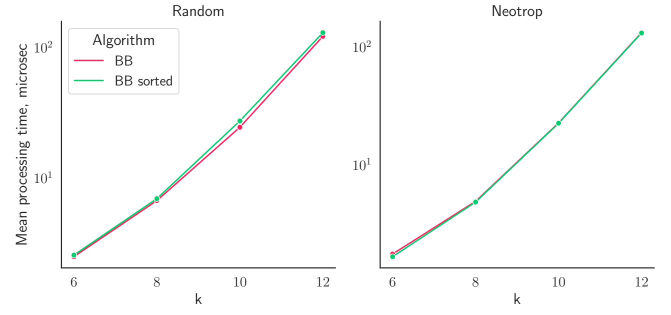


Fig. 9: Mean time (in microseconds, log-scale) to process a window for different values of $k$ by branch-and-bound on original data (BB) and on windows with sorted columns (BB sorted).

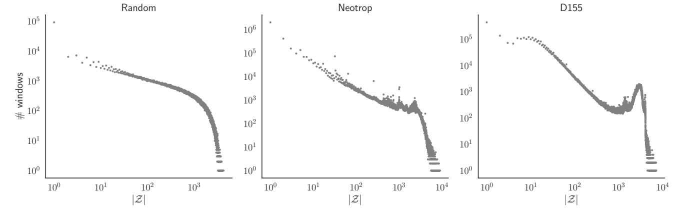


Fig. 10: Number of windows (y-axis) that have $|\mathcal{Z}|$ alive $k$-mers (x-axis) for the three datasets used in experiments.

Figure 9 presents the mean time spent by the branch-and-bound (BB) and its modified version (BB sorted) to perform phylo-$k$-mer computation for different values of $k$ on the *random* and the *neotrop* datasets. We could not obtain any significant gain in speed by sorting columns, unlike that in [5]. This is likely due to the fact that, for phylo-$k$-mer computation, we typically deal with a larger proportion of alive $k$-mers in a window than in PSSM applications.

# Appendix C
## Phylo-$k$-mer richness

Figure 10 plots the numbers of windows having different numbers of alive $k$-mers for $k = 10$ and $\varepsilon = (1.5/4)^k$. In other words, it shows the distribution of window "phylo-$k$-mer richness" for different datasets. Note that for real-world datasets (*neotrop*, *D155*), distributions have multiple peaks in the mid-range, unlike the one of the *random* dataset. This is due to the biological nature of these datasets: for this data, the probability distributions of the columns of the corresponding matrices $P^u$ are indeed almost guaranteed to be not uniform. This figure highlights that the impact of more phylo-$k$-mer-rich windows on the overall computation time is higher for real-world datasets than for simulated data.

# Appendix D
## Memory consumption

To evaluate and compare the memory requirements of the presented algorithms, we measured the peak RAM

| | BB | DC | DCCW |
|---|---|---|---|
| *Random* | 84.26 | 84.34 | 84.68 |
| *neotrop* | 1350.82 | 1351.19 | 1350.98 |
| *D155* | 1353.85 | 1353.80 | 1353.86 |

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.

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 $0.03\%$ 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 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 required by Definition 1.

## Appendix E
## Range product queries

Range product queries over column-maximum values can be solved using a classic technique for range sum queries over an array. First, we store maximum values of every column in an array $W_{max}$ of length $k$. Then, we compute the array

$$M[i] = \begin{cases} 1, & \text{if } i = 0 \\ \prod_{j=1}^{i} W_{max}[j], & 1 \le i \le k. \end{cases} \quad (2)$$

The answer to $M(j_1 : j_2)$ is then $M[j_2]/M[j_1 - 1]$.

## Appendix F
## Alternative thresholds

We include measurements of the running times of the presented algorithms for the values of thresholds different from $\varepsilon = (1.5/4)^k$ for the fixed value of $k = 10$. Figure 11 presents these measurements.

With the exception of Figure 11(a) for the *random* dataset, these plots confirm the observations made in the main text. In particular, BB only outperforms DC for phylo-$k$-mer-poor windows, and DCCW generally has the lowest running time. The plot for the *random* dataset and $\varepsilon = (1/4)^k$ is special: here the distribution of $|\mathcal{Z}|$ for different windows is very concentrated around its mean.

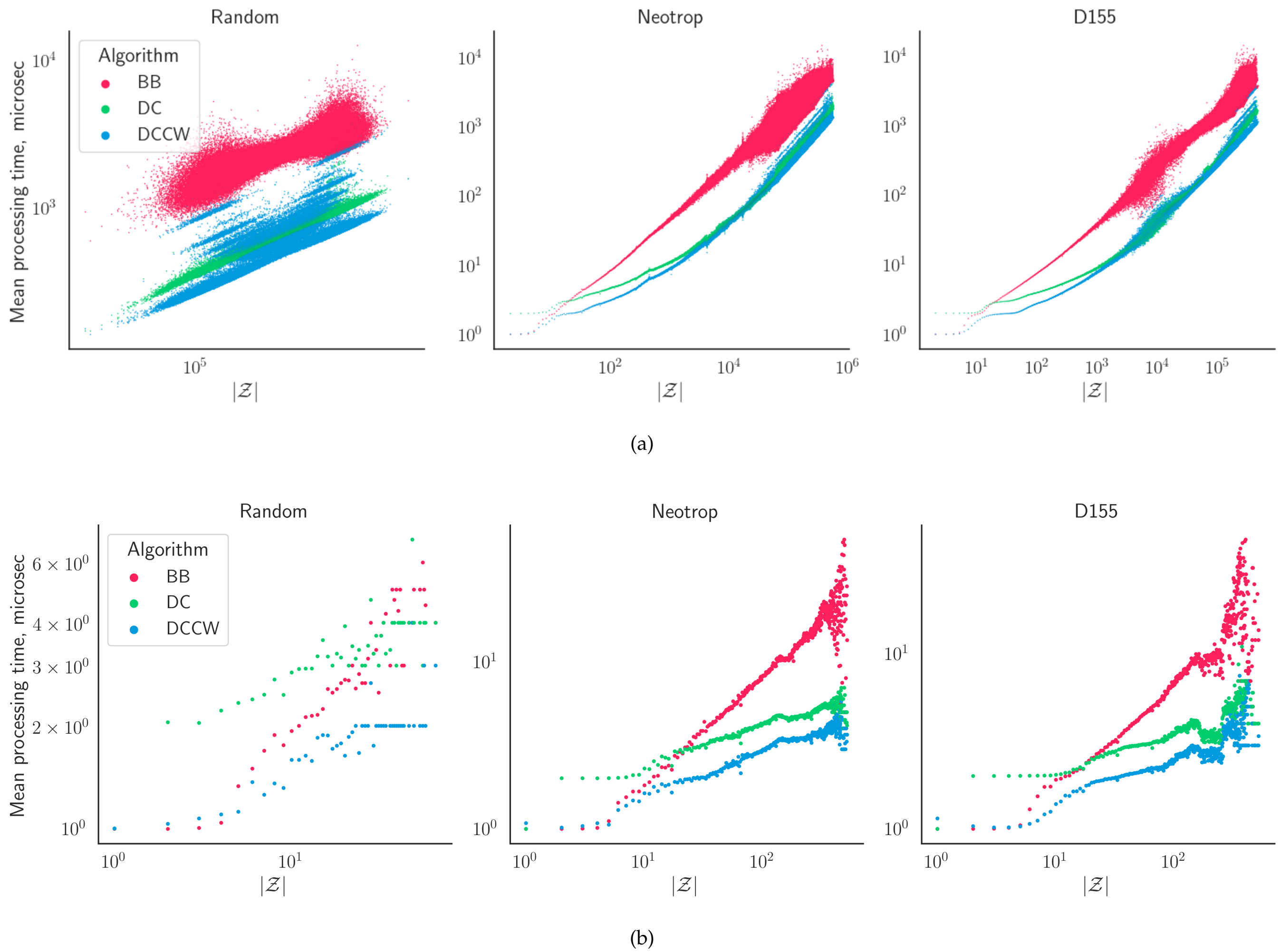


Fig. 11: Average time in microseconds to process a window of the alignment plotted against the number of phylo-$k$-mers alive for $k = 10$ and different values thresholds: (a) $\varepsilon = (1/4)^k$. (b) $\varepsilon = (2/4)^k$. Both axes are in log-scale.