Subsequence-Based Indices for Genome Sequence Analysis

Giovanni Buzzega[1]
Department of Computer Science, University of Pisa, Italy

Alessio Conte
Department of Computer Science, University of Pisa, Italy

Veronica Guerrini
Department of Computer Science, University of Pisa, Italy

Giulia Punzi
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy

Lorenzo Tattini
EURECOM, Biot, France
CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, Nice, France

Abstract

Compact indices are a fundamental tool in string analysis, even more so in bioinformatics, where genomic sequences can reach billions in length. This paper presents some recent results in which Roberto Grossi has been involved, showing how some of these indices do more than just efficiently represent data, but rather are able to bring out salient information within it, which can be exploited for their downstream analysis. Specifically, we first review a recently-introduced method [Guerrini et al., 2023] that employs the *Burrows-Wheeler Transform* to build reasonably accurate phylogenetic trees in an assembly-free scenario. We then describe a recent practical tool [Buzzega et al., 2025] for indexing *Maximal Common Subsequences* between strings, which can enable analysis of genomic sequence similarity. Experimentally, we show that the results produced by the one index are consistent with the expectations about the results of the other index.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases String Indices, Burrows-Wheeler Transform, Maximal Common Subsequences, Sequence Analysis, Phylogeny

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.20

Category Research

Related Version The algorithmic techniques shown in this paper are summarized from [13, 26].
Full Version: `https://doi.org/10.1186/s13015-025-00271-z` [13]
Full Version: `https://doi.org/10.1186/s13015-023-00232-4` [26]

Supplementary Material
Software (Source Code of PHY*BWT)*: `https://github.com/veronicaguerrini/phyBWT2` [26]
Software (Source Code of McDag*)*: `https://github.com/giovanni-buzzega/McDag` [13]

Funding *Giovanni Buzzega*: Partially supported by MUR PRIN 2022 project EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data (#2022TS4Y3N), and received funding from the European Union's Horizon 2020 Research and Innovation Staff Exchange programme under the Marie Skłodowska-Curie grant agreement No. 872539.

[1] corresponding author

© Giovanni Buzzega, Alessio Conte, Veronica Guerrini, Giulia Punzi, Giovanna Rosone, and Lorenzo Tattini;
licensed under Creative Commons License CC-BY 4.0

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.
Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter; Article No. 20; pp. 20:1–20:21
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Alessio Conte: Partially supported by MUR PRIN 2022 project EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data (#2022TS4Y3N).
Veronica Guerrini: Supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 Tuscany Health Ecosystem Spoke 6 CUP B63C2200068007 and I53C22000780001.
Giulia Punzi: Supported by the Italian Ministry of Research, under the complementary actions to the NRRP "Fit4MedRob - Fit for Medical Robotics" Grant (#PNC0000007), and received funding from the European Union's Horizon 2020 Research and Innovation Staff Exchange programme under the Marie Skłodowska-Curie grant agreement No. 872539.
Giovanna Rosone: Partially supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 Tuscany Health Ecosystem Spoke 6 CUP B63C2200068007 and I53C22000780001, and by project "Hub multidisciplinare e interregionale di ricerca e sperimentazione clinica per il contrasto alle pandemie e all'antibioticoresistenza (PAN-HUB)" funded by the Italian Ministry of Health (POS 2014–2020, project ID: T4-AN-07, CUP: I53C22001300001).

1 Introduction

Sequence analysis is one of the core branches of bioinformatics, and it is arguably one of the most fundamental tasks due to the abundance of genomic sequences enabled by advancements in sequencing technologies. Sequence analysis methods have a huge advantage compared to "in vitro" methods: once a dataset is available, it can be instantly and easily shared with anyone, it does not deteriorate or deplete, and can be analysed repeatedly with just regular computers, without the need of expensive ad-hoc machines. This is one of the reasons why much effort is dedicated to quickly produce new and more powerful sequencers, resulting in larger datasets available to the scientific community, ranging from bacteria and viruses to humans.

The size of genomic data can be daunting, as the length of genomic sequences ranges from thousands (for some viruses, or proteins) to millions (e.g., *E. Coli* genome) or billions (e.g., human genome). An important trade-off is immediately evident: more complex and refined approaches can extract better-quality information from the data, but require more computational resources to be executed, and may be not be applicable to complex organisms. The research community has been advancing in two main directions: developing more sophisticated algorithms, and extending their applicability to increasingly complex data. A key task in the latter direction is optimizing the way data is represented and handled, since just storing sequences in an uncompressed format may already require tens of GB of space. In this scenario, compact indices are extremely valuable tools, that typically employ algorithmic tricks to provide a good trade-off between the size of the index, and ease of access to the data (as well as support for specific queries).

In this paper, our aim is to give an overview of two recent research results concerning subsequence-based indices in bioinformatics, which involve Roberto Grossi both in their past development and in their current investigation of future directions. We experimentally highlight how these indices do more than just represent genomic data: their clever processing of the input enables the extraction of salient information that can be used for sequence analysis application tasks.

Specifically, the first research result we review is a method, called PHYBWT [25, 26], that addresses the problem of phylogenetic inference employing the Burrows-Wheeler Transform (BWT) [11]. The BWT is a text transformation with the remarkable feature of clustering together repeated sequences, a property originally intended to enhance compression, but now widely used in genome indexing algorithms [32]. Phylogenetic inference refers to the process of reconstructing the evolutionary relationships among species, or more generally, among taxa. The phyBWT methodology for phylogeny reconstruction uses the BWT of a string collection [7, 35], precisely of a group of sequences representing different taxa, to group related taxa together and to suggest evolutionary relationships. The main features of the phyBWT tool are its ability to work directly on raw sequencing data in an assembly-free scenario, and the fact that it does not rely on pairwise sequence comparisons, and thus on a distance matrix, but rather compares all the sequences simultaneously and efficiently.

The subsequent research result reviewed in this paper concerns the construction of a deterministic finite automaton (DFA) to efficiently index all *maximal common subsequences* (MCS) of two (or more) input strings. A common subsequence is a sequence of characters that occurs in the same order in all input strings, albeit not necessarily consecutively. An MCS is a common subsequence that is not a subsequence of any other common subsequence. The DFA, implemented as a labelled directed acyclic graph (DAG), is called McDag, and was presented in [12, 13]. Even if some previous works provide indices with better worst-case bounds [18, 28], the McDag index has been experimentally shown to be more efficient in practice for real-world genomic data, as it is significantly faster to build and typically smaller in size. Preliminary experiments (see Figure 3 from [12]) showed that the distribution of MCS lengths appears to behave differently when comparing very similar or dissimilar genomes.

Aiming to illustrate the potential of the two methods for genomic data analysis, we experimentally observe the MCS lengths distributions on genomic sequences within the same taxonomic group, to get a picture of how these distributions behave on closely related taxa versus more distant ones, using the phylogeny produced by phyBWT as a guide.

Outline of the paper

The paper is organized as follows. In the rest of the introduction, we will provide a review of the state of the art on both phyBWT and MCSs. Then, in Section 2 we will provide some technical preliminaries needed for the rest of the paper. Sections 3 and 4 will briefly review the main ideas behind the phyBWT [26] and McDag [13] works, respectively. Finally, Section 5 will present experiments on MCS lengths distributions for sequences from the same phylogenetic trees produced by phyBWT. The experimental comparison of phyBWT and McDag with their relative state of the art is out of the scope of the present work, and the interested reader can find it in their respective papers.

Related Work: Phylogenetic inference and PhyBWT

Sequence-based phylogeny aims to reconstruct evolutionary relationships between species, or more generally taxa, by comparing their DNA (or protein) sequences. The relationships among taxa are traditionally displayed in a tree-shaped diagram called *phylogenetic tree*, which can be rooted or unrooted. The leaves of the tree represent the contemporary organisms, while internal nodes represent common ancestors from which descendant lineages diverged.

The development of next-generation sequencing technologies in the early 2000s revolutionized this field, enabling researchers to sequence entire genomes quickly and cost-effectively. Such an amount of whole-genome sequencing data has lead to the need of advanced computational algorithms and tools for efficient phylogenetic inference.

Numerous sequence-based methods have emerged and evolved over time in this research field [43]. Most of them rely on a distance matrix by computing the pairwise evolutionary distances between every pair of input sequences representing the taxa. Distance measures are typically based on sequence alignment, and once the distances are obtained, the sequences are no longer utilized in the analysis.

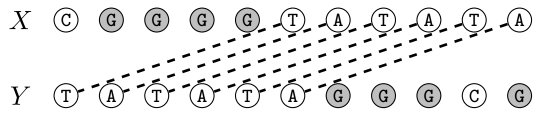


Figure 1 The two strings X, Y shown in the figure have as only LCS the string `TATATA` of length 6, shown with matching dashed edges. The MCS set is instead composed of three strings: `TATATA`, `GGGG`, and `CG`. In LCS-based analysis, the longer LCS prevents us from considering the second-longest MCS, `GGGG` (shaded), as a possible meaningful common pattern.

In 1992, Bandelt and Dress [4] introduced a technique called *split decomposition* that was shown to enhance phylogenetic analysis [5]. Based on a solid mathematical ground [4, 6], the split decomposition involves constructing a set of *splits* (binary partitions of the set of taxa) from a given dissimilarity matrix, each split being weighted by an isolation index that intuitively quantifies the strength of the split on the basis of the dissimilarity values. Given a distance matrix for ℓ taxa, the list of splits is computable in polynomial time (of order ℓ^6). Phylogenies in a tree-shaped form can be constructed by greedily selecting the splits with the highest isolation indices, as long as they are compatible[2]. Compatible splits correspond to a tree structure and, conversely, any tree can be represented by a set of compatible splits. Thus, ideal data gives rise to a phylogenetic tree, whereas phylogenetic networks, which generalize phylogenetic trees, are reconstructed when the splits are weakly compatible (see the tool SplitsTree [29]).

The increasing cost of the alignment task has led to the development of alignment-free approaches to efficiently quantify the dissimilarity between pairs of sequences [46]. Starting from the split decomposition idea, the authors of [44] introduced an alignment-free method called SANS that builds a list of splits and, from that, it infers the phylogeny by using SplitsTree. However, differently from the split decomposition theory, SANS builds the list of splits without relying on a distance matrix, and assigns weights to splits by counting fixed-length substrings shared among the sequences. According to [44], for ℓ taxa represented by sequences of length $\mathcal{O}(n)$ each, SANS runs in $\mathcal{O}(n\ell \log(n\ell))$ time.

The method phyBWT proposed in [26] and reviewed in this paper also belongs to the class of alignment-free approaches that infer phylogenetic relationships without relying on pairwise sequence comparisons. It differs from SANS as it does not build a list of splits, but rather defines a new strategy to draw a phylogenetic tree. Moreover, phyBWT evaluates sequence similarity/dissimilarity considering shared strings of varying length, without fixing a-priori the length of the common substrings. The interested reader can find a direct comparison between SANS and phyBWT in [26].

Related Work: Maximal Common Subsequences

Maximal common subsequences are a generalization of the well-known *Longest Common Subsequences* (LCSs), that is, common subsequences of maximum length: indeed, each LCS is, by definition, an MCS as well. LCSs are well-established in the context of genome sequence alignment [41], and the value of the length of an LCS can be used as a string similarity measure [8]. Still, by only considering the longest such sequences, (slightly) shorter but still relevant alignments might be discarded (see Figure 1). At the same time, going instead to the opposite extreme and considering *all* common subsequences would create too much

[2] A set of splits is compatible if, for every pair of splits, at least one of the four possible intersections between their parts is empty.

redundancy. A reasonable middle ground is thereby provided by MCSs, which can still be exponential in number (as LCSs can be too [24]), but are significantly fewer than all common subsequences.

The (shortest) MCS problem on strings[3] was proposed in [22], where the authors provided a dynamic programming algorithm for finding the shortest MCS, and other related problems. Sakai later provided the first (almost) linear-time algorithm to extract one MCS between two strings [39, 40]. This highlights a key difference from LCS computation, for which there exists a SETH-based quadratic conditional lower bound [1, 9]. When increasing the number of strings, this difference becomes even more pronounced: finding an LCS among an arbitrary number of strings is NP-hard [33], while there is a polynomial-time algorithm for extracting one MCS in the same setting [28]. MCSs were also recently employed as a tool for a parameterized LCS algorithm [10].

In the past years, several works have appeared on the topic of MCSs, more specifically in the direction of MCS enumeration and indexing, with Roberto Grossi contributing to many of them. Indeed, he took part in the first results concerning efficient MCS enumeration between two strings [16, 17], as well as in one of the two independent works that produced the first polynomial-sized indices for MCSs [18, 27]. Finally, as previously mentioned, he was involved in the development of the practical tool described and employed in the present paper [12]. While providing no formal theoretical bounds on the space complexity, experiments on genomic data have shown this algorithm to be more efficient in practice with respect to [18] (see the Experimental Analysis in [12, 13]). Worst-case complexity bounds for constructing McDag, and, more in general, for the minimum size of a DFA representing all MCSs, remain an open problem [13, Conclusions].

MCS problems have been studied for an arbitrary number m of input strings as well. Hirota and Sakai proposed a $O(nm \log n)$-time algorithm for computing one such MCS, where n is the total length of the input strings [28]. The practical indexing tool of Buzzega et al. [12] was also extended to deal with $m > 2$ strings in [13]. Moreover, the problem of efficiently indexing MCSs of an arbitrary number of strings, as well as their enumeration, has recently been shown to be unfeasible in time polynomial in the output size, unless P=NP [14].

2 Preliminaries

We consider a string $X = X[1] \dots X[|X|]$ as a sequence of characters from a finite and ordered alphabet Σ, where $X[i] \in \Sigma$ denotes the character at position i in X and $|X|$ denotes the total number of characters in X. Let $X[i, j]$ denote the *substring* $X[i] \dots X[j]$, for $1 \leq i \leq j \leq |X|$: a substring $X[i, |X|]$ (resp. $X[1, i]$) is called *suffix* (resp. *prefix*) of X, for any $1 \leq i \leq |X|$. We use special characters $\{\#, \$\}$ as markers delimiting input strings.

We say that string Z is a *subsequence* of X if there exist indices $1 \leq i_1 < \dots < i_{|Z|} \leq |X|$ such that $X[i_k] = Z[k]$ for $0 < k \leq |Z|$. If a subsequence can be mapped on X contiguously, i.e., for all $0 < k \leq |Z|$, $i_k = i_1 + k - 1$, then Z is a substring of X. Moreover, Z is a *common subsequence* of strings X and Y if Z is a subsequence of both X and Y(see Figure 1). More specifically, let us call a pair (i, j) a *match* when $X[i] = Y[j]$: letting $1 \leq j_1 < \dots < j_{|Z|} \leq |Y|$ be the indices such that $Y[j_k] = Z[k]$ for $0 < k \leq |Z|$, then each pair (i_k, j_k) is a match (as $X[i_k] = Y[j_k]$) and we say that the pairs $(i_1, j_1), \dots, (i_{|Z|}, j_{|Z|})$ form a *matching* in X and Y, whose corresponding string is Z. We observe that matches induce a partial order, defined

[3] The concept of MCS in a more general form actually first appeared in data mining applications [3], where they were defined over ordered sequences of *itemsets*, instead of over strings. The string setting we consider for MCS can be seen as a special case of this framework, where each itemset is a singleton.

as $(i, j) < (i', j')$ iff $i < i'$ and $j < j'$, which is total if the pairs belong to the same matching; $(i, j) \leq (i', j')$ is analogously defined. A string Z is a *maximal common subsequence* (MCS) of X and Y if there is no string $W \neq Z$ that satisfies both conditions: *(i)* W is a common subsequence of X and Y, and *(ii)* Z is a subsequence of W. The set of all strings that are maximal common subsequences is denoted by $MCS(X, Y)$.

We next introduce some graph notions. A *directed graph* $G = (V, E)$ consists of a set of nodes V and a set of edges $E \subseteq V \times V$, where each edge (u, v) is an ordered pair of nodes that specifies a direction from u to v. Two edges (u, v) and (w, z) are said to be adjacent if $v = w$. A path in G is a sequence of distinct edges, each adjacent to the next. If the path starts at node s and ends at node t, it is called an st-path; it is a cycle when $s = t$. A *DAG* is a directed acyclic graph. Given a node u, the set $N^+(u)$ indicates the out-neighbor nodes v such that $(u, v) \in E$, and the set $N^-(u)$ indicates the in-neighbor nodes v such that $(v, u) \in E$. The out-degree of u is $d^+(u) = |N^+(u)|$, and its in-degree is $d^-(u) = |N^-(u)|$; u is a *source* if $d^-(u) = 0$, and a *sink* if $d^+(u) = 0$. In a *labeled* DAG $G = (V, E, l)$ each node u is associated with a character $l(u) \in \Sigma \cup \{\#, \$\}$. In Section 4 we also consider labeled DAGs in which each node u is associated with a match $m(u) = (i_u, j_u)$.

3 BWT-based Phylogenetic Inference

In this section we review phyBWT, a methodology first introduced in [25] and subsequently refined in [26], for reconstructing a phylogenetic tree bypassing the standard computationally expensive steps of sequence alignment and *de novo* assembly. It can take as input any type of sequence data representing taxa, such as whole-genome sequences and raw sequencing reads.

The approach exploits the inherent combinatorial properties of the extended Burrows-Wheeler Transform (eBWT) [7, 35] to index and detect relevant common substrings of varying length. The common substrings then play a crucial role in building partition trees without performing pairwise comparisons between sequences. This is the primary feature of phyBWT: the tree structure is inferred by comparing all the sequences simultaneously and efficiently, without resorting to a distance matrix.

The second remarkable feature of phyBWT is that, to the best of our knowledge, it is the first approach to apply the properties of the eBWT to the idea of decomposition for phylogenetic inference. By indexing the sequences in the eBWT, we can identify maximal common substrings of varying lengths that are used to group sequences together and to partition groups of taxa based on their shared substrings.

Finally, the worst-case running time of phyBWT is $O(N\ell)$, where ℓ is the number of taxa and N is the total length of all the taxa sequences, using $O(N + \ell^2)$ space.

In the following, we first provide an overview of the preliminary notions. These include the extended Burrows-Wheeler Transform employed for detecting common substrings and the *positional clustering* framework, which overcomes the limitation of a priori fixing the length of the common substrings. Subsequently, we delineate the tree reconstruction methodology of phyBWT according to [26].

3.1 Burrows-Wheeler transform and Common Substrings

The Burrows-Wheeler Transform (BWT) [11] is a well-known reversible transformation that permutes the symbols of a string in such a way that, as a result, the runs of equal symbols tend to increase. In addition to enhancing the performance of memoryless compressors, the BWT plays a crucial role in the development of efficient self-indexing compressed data structures.

The BWT was first extended to a string collection in [35] (eBWT) by sorting cyclic rotations of all the strings in the collection according to a special order, called ω-order. In order to use the lexicographic order rather then the ω-order, in [7], a variant of the eBWT was defined by appending a distinct end-marker symbol to each string and lexicographically sorting the suffixes of all the strings in the collection. In the following, we call ebwt the output string[4] defined in [7], and introduce some auxiliary data structures that allow to detect common substrings in a string collection.

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be a collection of m strings. We assume that each string $s_i \in \mathcal{S}$ is a sequence of $n_i - 1$ characters from Σ followed by a special end-marker symbol $\$_i$, i.e. $s_i[n_i] = \$_i$, which is lexicographically smaller than any other symbol in Σ and $\$_i < \$_j$, if $i < j$. The total number of characters in $\mathcal{S}$ is $N = \sum_{i=1}^{m} n_i$. The $\mathsf{ebwt}(\mathcal{S})$ string is defined by concatenating the symbols preceding each suffix of the lexicographically sorted list of suffixes of all the strings $s_1, \ldots, s_m$, where each s_i is circular. The *longest common prefix* (LCP) array [34] of $\mathcal{S}$ (denoted by $\mathsf{lcp}(\mathcal{S})$) is the array of length N storing the length of the longest common prefix between any two consecutive suffixes in lexicographically sorted list of suffixes of $s_1, \ldots, s_m$, using the convention that $\mathsf{lcp}(\mathcal{S})[1] = 0$.

Finally, if $\mathcal{S}$ comes in ℓ parts, namely $\mathcal{S} = S_1 \cup S_2 \cup \ldots \cup S_\ell$, where each S_i is a non-empty subset of $\{s_1, \ldots, s_m\}$, and all the subsets are pairwise disjoint, then the *color document array* of $\mathcal{S}$ (denoted by $\mathsf{cda}(\mathcal{S})$) is the array of length N storing the indices of the subsets to which the $\mathsf{ebwt}(\mathcal{S})$ symbols belong. The set $\mathcal{S}$ is omitted if it is clear from the context.

▶ Remark 1. Let $\mathcal{R} \subset \mathcal{S}$. The data structures $\mathsf{ebwt}(\mathcal{R})$, $\mathsf{lcp}(\mathcal{R})$, and $\mathsf{cda}(\mathcal{R})$ can be deduced through a linear scan of the larger $\mathsf{ebwt}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$, and $\mathsf{cda}(\mathcal{S})$, as the relative order of suffixes holds (see [7], cf. also [15]).

One technique to find common substrings of fixed length k in $\mathcal{S}$ is based on the use of LCP-intervals. An *LCP-interval* of lcp-value k is a maximal interval $[i, j]$ such that $\mathsf{lcp}[r] \geq k$ for $i < r \leq j$ (defined slightly differently from [2]); in other words, the interval $[i, j]$ corresponds to suffixes in the lexicographically sorted list that share at least the first k characters. Nevertheless, the length of the common prefix in any LCP-interval could be longer than k, possibly revealing common substrings of greater length.

To overcome the limitation of a priori fixing the length of common substrings in $\mathcal{S}$, the authors of [37] introduced a framework called *positional clustering*. According to this framework, the boundaries of the intervals in the LCP array are data-driven, and not established a-priori by a fixed k. Specifically, the intervals of interest are those enclosed between two "local minima" in the LCP array. In fact, intuitively, a local minimum in the LCP array indicates a shortening of the common prefix. Moreover, to exclude intervals associated with short random prefixes, a minimum prefix length k_m can be established. Formally, an *eBWT positional cluster* is a maximal substring $\mathsf{ebwt}[i, j]$ such that $\mathsf{lcp}[r] \geq k_m$, for all $i < r \leq j$, and none of the indices $i < r \leq j$ is a *local minimum* of the LCP array[5]. By definition, we have that any two different eBWT positional clusters are disjoint.

▶ Remark 2. Each eBWT positional cluster $\mathsf{ebwt}[i, j]$ corresponds to suffixes in the lexicographically sorted list that have a common prefix (i.e., a common substring) of length given by the minimum between $\mathsf{lcp}[i + 1]$ and $\mathsf{lcp}[j]$ (see [37, Theorem 3.3]). Thus, each eBWT positional cluster $\mathsf{ebwt}[i, j]$ corresponds to a substring in $\mathcal{S}$, and the values in $\mathsf{cda}[i, j]$ provides the information about the strings that contain it.

[4] In the literature, the extended transform of [7] is also called multi-string BWT [21] or mdolEBWT [15].

[5] According to [26], an index r is said a local minimum if $\mathsf{lcp}[r-1] > \mathsf{lcp}[r]$ and $\mathsf{lcp}[r] < \mathsf{lcp}[r+s]$, where $s > 1$ is the number of adjacent occurrences of $\mathsf{lcp}[r]$ from position r. For instance, the local minima of $\mathsf{lcp} = [2, 1, 3, 5, 4, 4, 2, 2, 7]$ are indices 2 and 7, corresponding to LCP values of 1 and 2, respectively.

Remarks 1 and 2 are key to the phyBWT method, which detects common variable-length substrings of a subset of taxa and uses this information to reconstruct a phylogenetic tree.

3.2 Tree Reconstruction Method

The methodology proposed in [26] reconstructs a tree T through a series of refinement steps performed on groups of taxa.

Formally, we denote the set of leaves as $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ where each S_i corresponds to a taxon, which could be represented by a single sequence (e.g., genome sequence) or a string collection (e.g., sequencing reads).

The tree T is defined as a *partition tree* of the set $\mathcal{S}$:

- each node of T corresponds to a non-empty set of taxa $S' \subseteq \mathcal{S}$;
- the root of T corresponds to $\mathcal{S}$;
- each leaf of T corresponds to a distinct taxon $S_i \in \mathcal{S}$, and vice-versa;
- for each node corresponding to S', its children form a partition of S'.

We define the operation of *adding a node* to T by a set: a set $S' \subseteq \mathcal{S}$ can be added to T only if it is *compatible*, i.e., if every other node of T corresponds to a set S'' that satisfies one of these conditions: $S'' \subset S'$, $S'' \supset S'$, or $S'' \cap S' = \emptyset$ (i.e. no partial overlap between S'' and S'). If this is the case, there is only one way to add S' to T, namely, S' becomes a child of the smallest set $P \supset S'$ of T (by cardinality), and all the other children of P that are contained in S' become the children of S'. The resulting T is still a partition tree.

We describe the method by first explaining the tree reconstruction procedure, which applies a refinement procedure iteratively, and then by briefly sketching the inner refinement algorithm. The rationale of the refinement algorithm is to group together nodes of T whose associated sequences share variable-length substrings not found in other sequences, and to interpret this fact as a common feature of the group that differentiates it from the others.

Tree reconstruction via the refinement procedure. The key idea is to refine an intermediate partition tree by taking one of its internal nodes and applying the refinement procedure to the groups of taxa corresponding to its children. Here, we consider refinement as a blackbox that uses the eBWT and its related data structures to produce a list of compatible subsets, which are new nodes that can be added to the partition tree. This process allows for fine-grained node clustering, by restricting the input data to the sequences of the relevant subtree. This is repeated until all internal nodes in the partition tree have only two children, or no more refinements are possible. We report the pseudocode in Algorithm 1.

The tree produced is an unrooted tree, but for the sake of simplicity, we describe it as rooted. At the beginning the unrefined partition tree T (Line 1 in Algorithm 1) is a rooted star that has $\ell + 1$ nodes: root $\mathcal{S}$ (non-final) and children $S_1, \ldots, S_\ell$ marked as final. The mark *final* for a node indicates that no more refinement is possible at that node.

The algorithm iteratively processes any non-final node X of T (Line 3): given the list of nodes $C_1, \ldots, C_h$ that are children of X, refinement (Line 6) returns a list $L = L_1, \ldots, L_s$, with $s < h$, of compatible subsets of $\bigcup_k C_k$. The draw_and_mark function adds the nodes (possibly non-final) listed in L to T (Line 7).

To mark the node X as final, the draw_and_mark function checks if L is empty (Line 10). If L is not empty, a new internal node of T is created for each $L_i \in L$. Any inserted node, as well as X, is marked final, if it has only two children; otherwise, it needs to be further refined and is added to the queue (Lines 14-16). One possible iteration of Algorithm 1 can be found in [26, Figure 2].

Algorithm 1 Iterative refinement of the partition tree (Algorithm 1 from [26]).

input : $\ell, \mathsf{ebwt}(\mathcal{S}), \mathsf{lcp}(\mathcal{S}), \mathsf{cda}(\mathcal{S})$
output : A tree whose leaves are colored with $1 \ldots \ell$, each color being a taxon of $\mathcal{S}$

Let $T \leftarrow$ Rooted star with a non-final root $\mathcal{S}$, and final leaves colored $1 \ldots \ell$
Queue.push($\mathcal{S}$)
while Queue *is not empty* **do**
  $X \leftarrow$ Queue.pop()
  $C_1, \ldots, C_h \leftarrow X$.children()
  $L \leftarrow$ REFINEMENT($\mathsf{ebwt}(\mathcal{S}), \mathsf{lcp}(\mathcal{S}), \mathsf{cda}(\mathcal{S}), \{C_1, \ldots, C_h\}$)
  DRAW_AND_MARK(T, L, X, Queue)
Function *DRAW_AND_MARK* (T, L, X, Queue)
  if L *is empty* **then**
    Mark X in T as final `// cannot further refine` X
  else
    foreach *set* L_i *of* L **do**
      Add L_i as a node in T if not already present
    Mark as final every new node with two children in T
    Add to Queue all new nodes not marked final
    Mark X as final if it has two children, otherwise add it to Queue
Return T

The refinement procedure. The inner REFINEMENT function returns a list L of compatible subsets starting from a set of sibling nodes $C_1, \ldots, C_h$ of T, which correspond to some (not necessarily all) taxa. Specifically, if C_i is a leaf, then C_i corresponds to one taxon (represented by a single sequence or a string collection), otherwise C_i is an internal node and corresponds to the subset $\mathcal{R}_i \subset \mathcal{S}$ comprising all the taxa associated with the leaves of the subtree rooted at C_i. Let $\mathcal{R} = \bigcup_{i=1}^{h} \mathcal{R}_i$ be the set of all the taxa corresponding to nodes $C_1, \ldots, C_h$.

To quantify the similarity of a subset of taxa in $\mathcal{R}$ in terms of their common substrings, the eBWT positional clustering framework is employed and scores are assigned to some of the eBWT positional clusters detected. More precisely, given $\mathcal{R} \subset \mathcal{S}$, by Remark 1, we linearly scan the data structures $\mathsf{ebwt}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$ and $\mathsf{cda}(\mathcal{S})$ to detect and analyse eBWT positional clusters in $\mathsf{ebwt}(\mathcal{R})$. Among all the eBWT positional clusters detected, by Remark 2, we consider *relevant* the ones associated with common substrings that are shared by a sufficiently large number of taxa in $\mathcal{R}$ (but not by all of them) and cannot be extended on the left – the reader can find details in [26, Definition 3.5]. Since any relevant positional cluster is associated with a unique subset $R \subset \mathcal{R}$ of taxa sharing a common substring of variable length, we assign the length of that common substring as the cluster's score for subset R (see also Remark 2). After analysing all the relevant positional clusters, we have a weighted list $\mathcal{L}$ of subsets of $\mathcal{R}$. Each subset R in $\mathcal{L}$ corresponds to *at least one* relevant positional cluster, and its weight is the sum of all the scores for R over all the relevant positional clusters.

Finally, to build up the output list L, we sort the subsets in $\mathcal{L}$ by their weight and greedily select those with the highest weight that are compatible with each other. For computational efficiency, we stop the greedy procedure after a certain number of consecutive unsuccessful attempts to add elements to L (more details in [26]).

4 The McDag Compact Index for Maximal Common Subsequences

In this section we describe the main ideas of the practically efficient compact index McDag introduced in [12]. In the present paper, for the sake of simplicity, we describe the results for two input strings, even if they have also been extended to handle an arbitrary number of strings in [13].

Let us first formalize the definition of an MCS index as follows:

▶ **Definition 3** ([12], Section 2.2). *Given two strings X and Y of length $O(n)$, a labeled DAG $G = (V, E, l)$ is an index for $MCS(X, Y)$ if the following conditions hold:*
1. *Each node u (other than source or sink) is associated with a match denoted as $m(u) = (i, j)$, and has label $l(u) = X[i] = Y[j]$, where $1 \leq i \leq |X|$ and $1 \leq j \leq |Y|$.*
2. *There exist a single source s and a single sink t, with special values $m(s) = (0, 0)$, $l(s) = \#$, and $m(t) = (|X| + 1, |Y| + 1)$, $l(t) = \$$.*
3. *Each st-path $P = s, x_1, ..., x_h, t$ is associated with unique string $Z = l(x_1), ..., l(x_h) \in MCS(X, Y)$, and the associated matching for P must satisfy $m(x_1) < \cdots < m(x_h)$.*
4. *For each $Z \in MCS(X, Y)$ there is a corresponding st-path $P = s, x_1, ..., x_h, t$ such that $Z = l(x_1), ..., l(x_h)$.*

Let us note that a naive construction of such an MCS index (e.g., through a trie) could potentially require exponential time and space, as the number of nodes may be proportional to the number of MCS, which can in turn be exponential in the input size. Constructing such a compact MCS index in an efficient way is therefore not trivial.

We start by giving a high-level idea of how this problem is solved by McDag in Section 4.1, and then, in Section 4.2, we focus on explaining how to efficiently compute the frequency distribution of MCS lengths from the McDag index.

4.1 Overview of McDag Construction

The best way to define McDag is to employ a two-phase scheme. In the first phase, an *approximate rightmost co-deterministic index* $A = (V_A, E_A, l_A)$ for the set of MCSs is built. Approximate, rightmost, and co-deterministic respectively mean that (i) A indexes both the whole set of MCSs as well as some non-maximal common subsequences; (ii) for each edge (v, u) no character $l_A(v)$ appears between the positions defined by matches $m(v)$ and $m(u)$; and (iii) each node of A has at most one in-neighbor labeled with any character $c \in \Sigma$. Then, the second phase builds a *deterministic* version of A (i.e., where each node has no more than one out-neighbor per character) that does not contain any non-maximal common subsequence, yielding the final McDag. This latter procedure is called McConstruct, and its pseudocode is reported in Algorithm 2.

Empirical results show that the size of the initial approximate index A plays an important role in determining the size of the output MCS index. For this reason, we here review a method to construct A that tries to include few non-maximal common subsequences to begin with. Nevertheless, McConstruct correctly produces an MCS index for any input approximate rightmost co-deterministic index. For example, one could use a variant of the Common Subsequence Automaton [19, 20, 42], which models all common subsequences of a set of input strings.

First phase. We start by building a deterministic approximate MCS index $D = (V_D, E_D, l_D)$: we first add a source s_D with associated match $m(s_D) = (0, 0)$, corresponding to character $l(s_D) = \#$; then, we start to visit all nodes u in V_D. Throughout construction we ensure that all nodes have distinct matches: if $m(u) = m(v)$ then $u = v$. When visiting node u, we consider for each character $c \in \Sigma$ the closest match $(i_c, j_c) > m(u)$. If there exists a match m' such that $m(u) < m' < (i_c, j_c)$, then we discard (i_c, j_c). Otherwise, we identify node v with match $m(v) = (i_c, j_c)$, or create v if it is not present. Then, we connect node u to node v. If we are not able to connect u to any node, we connect it to the sink t_D, which has match $m(t_D) = (|X| + 1, |Y| + 1)$ and label $l(t_D) = \$$.

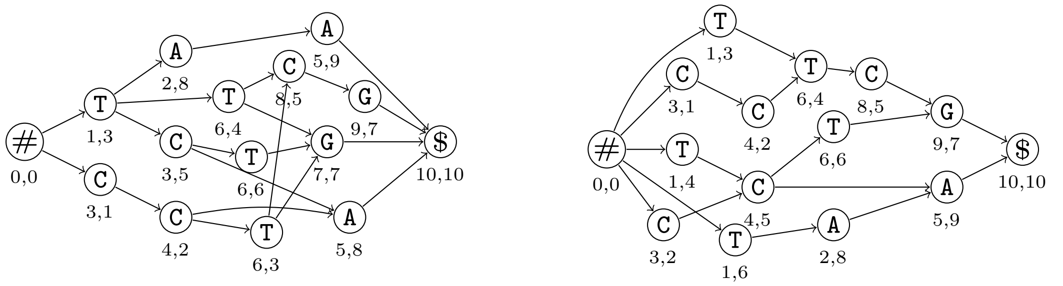
To build A, we repeat the same process in the opposite direction, reading the input strings right-to-left while using D as a guide: for each node u, we define a corresponding set of nodes $F(u) \subseteq V_D$ as the nodes that share a suffix with u. We start by adding t_A, with $F(t_A) = \{t_D\}$ and match $m(t_A) = m(t_D)$. To ensure that $F(u)$ is completely defined, we visit u only after we have visited all its potential out-neighbors, i.e., all nodes w that have match $m(u) < m(w)$. When processing a node u, we add its in-neighbors and enrich their $F(\cdot)$ sets as follows. We consider each node $x \in V_D$ such that $(x, y) \in E_D$ for some $y \in F(u)$, and if there is a match m' such that $m(x) < m' < m(u)$, we discard x. For each character c associated with the non-discarded nodes x, we select node v with match $m(v) = (i_c, j_c)$ such that c does not appear in $X[i_c + 1] \ldots X[i_u - 1]$ and $Y[j_c + 1] \ldots Y[j_u - 1]$, or create it if not present, and add it as an in-neighbor of u. We then add all non-discarded nodes x to $F(v)$.

Algorithm 2 McConstruct (Algorithm 1 from [12]).

Data: Input $A = (V_A, E_A, l_A)$: rightmost approximate co-deterministic MCS index with source s_A
Result: A deterministic MCS index $G = (V, E, l)$

Initialize $G = (V, E, l)$, where $V = \{s\}$, $E = \emptyset$, $m(s) = m(s_A)$, $l(s) = \#$
    `//` $F(u)$ `is the set of nodes in` A `corresponding to` u `in` G
$F(s) \leftarrow \{s_A\}$
while *there exists* $u \in V$ *with no out-neighbors and* $l(u) \neq \$$ **do**
    Initialize $N_c = \emptyset$ for all $c \in \Sigma \cup \{\$\}$
    forall $(x, y) \in E_A$ *such that* $x \in F(u)$ **do**
        Add y to N_c, where $c = l(y)$
    Initialize $P = \emptyset$
    forall $N_c \neq \emptyset$ **do**
        $i_c \leftarrow \min\{i_z \mid (i_z, j_z) = m(z) \wedge z \in N_c\}$
        $j_c \leftarrow \min\{j_z \mid (i_z, j_z) = m(z) \wedge z \in N_c\}$
        Add match (i_c, j_c) to P
    forall $N_c \neq \emptyset$ *and* $p \in P$ **do**
        Remove all y from N_c such that $p < m(y)$
    forall $N_c \neq \emptyset$ **do**
        if *no node* $w \in V$ *has* $F(w) = N_c$ **then**
            Add new node w to V
            Set $F(w) = N_c$, $m(w) = (i_c, j_c)$, $l(w) = c$
        else Let $w \in V$ be the node such that $F(w) = N_c$
        Add edge (u, w) to E
return $G = (V, E, l)$

Second phase. Given $A = (V_A, E_A, l_A)$ with source s_A from the first phase, we apply Algorithm 2 (McConstruct) to obtain a graph $G = (V, E, l)$ that becomes our McDag with source s. Again, we associate each node $u \in V$ with a set $F(u)$ of nodes from V_A, all having the same label as u (initially, $F(s) = \{s_A\}$ with label $\#$). This time, a node



(a) The first deterministic approximate index D, with $|V_D| = 15$ and $|E_D| = 21$.

(b) The co-deterministic approximate index A, with $|V_A| = 15$, and $|E_A| = 19$.

Figure 2 First phase of McDag construction for input strings X = `TACCATGCG` and Y = `CCTTCTGAA`.

$x \in F(u)$ must share at least one prefix with u. At each step we take a node $u \neq t$ and add its out-neighbors. To do so, we take the out-neighbors of x in A and filter-out the ones whose matches are to the right of some match $(i_c, j_c) > m(u)$, as they cannot lead to an MCS: (i_c, j_c) is a witness to defy their maximality. Then, we identify a node v with that same associated set of filtered out-neighbors $F(v)$, or we add it if not present, and add edge (u, v) to G. The key difference is that in the first phase each node u is uniquely identified by match $m(u)$, while in the second phase it is uniquely identified by the set $F(u)$. We end up having a single sink t, corresponding to \$, only occurring at the end of both strings.

In the rest of this section we illustrate the described method with an example, and we highlight the key features that make McConstruct work. Consider the two input strings X = `TACCATGCG` and Y = `CCTTCTGAA`. Figure 2 depicts the indices constructed during the first phase. More specifically, Figure 2a depicts the deterministic approximate index D, built by reading both strings left-to-right, while Figure 2b shows the co-deterministic approximate index A which is constructed using D. Upon careful inspection, one can observe that A does not contain the non-maximal sequence `TTG` as a source-to-sink path, whereas D does. However, A still contains `CCTG`, which is also not maximal because of `CCTCG`.

Non-maximal common subsequences such as `TTG` or `CCTG` can be characterized in both of these data structures by using the concept of *subsequence bubbles*. A subsequence bubble is formed by a pair of paths that start and end at common nodes but are otherwise node-distinct, with the added condition that the shorter path spells a subsequence of the longer one. If a non-maximal common subsequence is contained in the index, the corresponding st-path must pass through the shorter path of at least one subsequence bubble. For instance, in Figure 2a a subsequence bubble is given by the pair of paths $(1,3),(6,4),(7,7)$ and $(1,3),(3,5),(6,6),(7,7)$, in which the short path spells `TTG` and the long path spells `TCTG`, thus witnessing the non-maximality of `TTG`. The main goal of McConstruct is to ensure that the resulting DAG contains no such subsequence bubbles. We refer the interested reader to [12] for the proof of correctness.

Figure 3 finally shows the McDag index, resulting from using the index A as input for McConstruct. Empirically, McDag has been shown to usually produce smaller indices with respect to the provably polinomially-bounded index of [18]. Despite this, finding a polynomial theoretical bound on the size of McDag remains an open problem. Note that here we are comparing the size of the indices as output by the respective construction methods, without applying any node removal. Otherwise, a simple automaton minimization

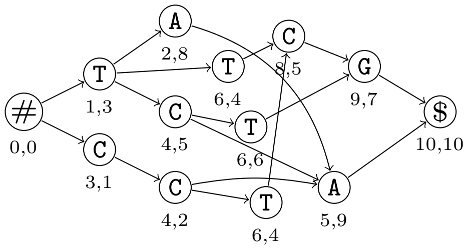


Figure 3 The McDag index for input strings $X = \texttt{TACCATGCG}$ and $Y = \texttt{CCTTCTGAA}$ has $|V| = 13$, $|E| = 17$ ($F(\cdot)$ sets omitted for compactness). Note that there are two nodes for match $(6, 4)$: this necessarily means that the two nodes have different $F(\cdot)$ sets.

algorithm (such as Revuz's algorithm [38] for acyclic deterministic finite automata) could reduce the number of nodes to a minimum, independently of the starting MCS index. Finding a tight bound on the size of the resulting minimal index also remains an open problem.

4.2 Generation of MCS Lengths Distribution

Building a deterministic index for MCSs allows us to perform a number of interesting operations such as enumeration, counting, and random access. One particular operation we might be interested in is related to counting: in Section 5 we will show experiments comparing the distribution of the MCS lengths for different pairs of genomic sequences. Here, we explain how one can generate such a distribution, by counting the number of paths for each path-length inside a DAG, using dynamic programming. The code for generating this distribution has been made available with the original paper [12], but the underlying algorithm was not detailed therein. We give a brief description here.

Consider a DAG $G = (V, E)$ and a generic node $u \in V$. Let $d(u)_i$ be the number of paths of length i that start from node u and end in a sink. These values can be computed as follows. For any sink t we define $d(t)_0 = 1$. Then, for the remaining nodes we can compute $d(u)_{i+1} = \sum_{(u,v)\in E} d(v)_i$, for all $i > 0$. Indeed, if all out-neighbors v of u have already computed their $d(v)_i$ values, node u can gather the sum of paths of length i and set the result as the number of paths of length $i + 1$ starting from u. At the end of the procedure, the distribution of the path lengths will be stored at the sources of the DAG.

The main problem of the procedure we just described is that every counter $d(u)_i$ can take non-negligible space and may not fit into a machine word. As previously mentioned, the number of MCSs can be exponential in the length of the input strings. This in turn means that the space required to store the number of paths of a given length is $O(n)$ bits. Since for our purposes we are interested in the *qualitative* distribution of the MCS lengths, we can use a trick (commonly known as log-sum-exp) to ensure that each $d(u)_i$ value can fit into a machine word. Namely, instead of storing the number of paths in $d(u)_i$, we store the logarithm of that number, as $d^*(u)_i = \log(d(u)_i) = O(n)$ for all u and i. To directly compute the value $d^*(u)_{i+1}$ we do the following: first, we find the maximum number of paths of length i among all out-neighbors as $\alpha_i = \max_{(u,v)\in E} d^*(v)_i$; then we compute $d^*(u)_{i+1} = \alpha_i + \log\left(\sum_{(u,v)\in E} 2^{(d^*(v)_i - \alpha_i)}\right)$.

5 Experiments

In this section, we experimentally review that the BWT-based tool phyBWT can achieve benchmark-level accuracy in phylogenetic reconstruction by exploiting the common substrings among taxa. Furthermore, we provide experimental evidence suggesting that the tool for MCSs indexing could offer valuable insights for inferring evolutionary relationships.

Specifically, the plots showing the distribution of the MCS lengths reveal a notable correlation between sequences associated with taxa that are close in the phylogenetic tree.

Datasets. For this study, we selected two datasets from two well-known viruses: the Human immunodeficiency virus (HIV) and the Ebola virus. Since viruses can evolve rapidly, viral phylogenies are challenging and often look very different. However, clade classification plays a crucial role in virology, since each clade (or subtype) represents a group with shared genetic similarities.

The HIV dataset comprises 43 HIV-1 complete genomes that have been used in the literature [45]. Thirty-five sequences belong to the major group (Group M) which is divided into subtypes A, B, C, D, F, G, H, J, K; seven sequences are from the minor Groups N and O, and one CPZ sequence (CIV strain AF447763) is an outgroup. The average length of the sequences is 9267 base pairs. The reference sequences have been carefully selected in [31] according to several criteria, and can be downloaded from the Los Alamos National Laboratory HIV Sequence Database[6].

The Ebola dataset comprises 20 published sequences from [23] selected in [30]. The Ebolavirus genus includes five viral species: Ebola virus (*Zaire ebolavirus*, EBOV), *Sudan virus* (SUDV), *Tai Forest virus* (TAFV), *Bundibugyo virus* (BDBV), and *Reston virus* (RESTV). The average length of the sequences is 18900 base pairs.

Phylogeny reconstruction. For the HIV dataset, Figure 4 depicts the phylogeny produced by phyBWT in [26]. Resembling the benchmark phylogeny depicted in [45, Fig. 2], subtypes are distinctly grouped together in different branches: subtypes B and D (resp. C and H) are closer to each other than to the others, and subtype F (resp. A) contains two distinguishable sub-subtypes F1 and F2 (resp. A1 and A2) that are closely related to subtypes K and J (resp. G), while subtypes N and O are external.

For the Ebola dataset, Figure 5 depicts the phylogeny produced by phyBWT in [26]. According to the benchmark phylogeny depicted in [30, Fig. 4], phyBWT exactly separated the five species. The EBOV sequences are clustered into a monophyletic clade, and BDBV and TAFV viruses are positioned close and then clustered with the EBOV branch. The SUDV clade is placed as sister to the EBOV, TAFV and BDBV clade, like in [30, Fig. 4E].

Given the required data structures, phyBWT reconstructs the proposed phylogeny for each dataset in less than one second by performing only two iterations of Algorithm 1 for the Ebola dataset and three iterations for the HIV dataset, with a RAM usage of approximately 8.5 MB.

MCS length distribution. We report in Figures 6 and 7 the logarithmic distribution of the MCS lengths of different viruses taken from the HIV and Ebola datasets. On the x-axis we find the various lengths of the MCSs, while on the y-axis the logarithm of their quantity. Specifically, Figure 6 considers the four taxa *AF286238 A2, AF286237 A2, U51190 A1,*

[6] http://www.hiv.lanl.gov/

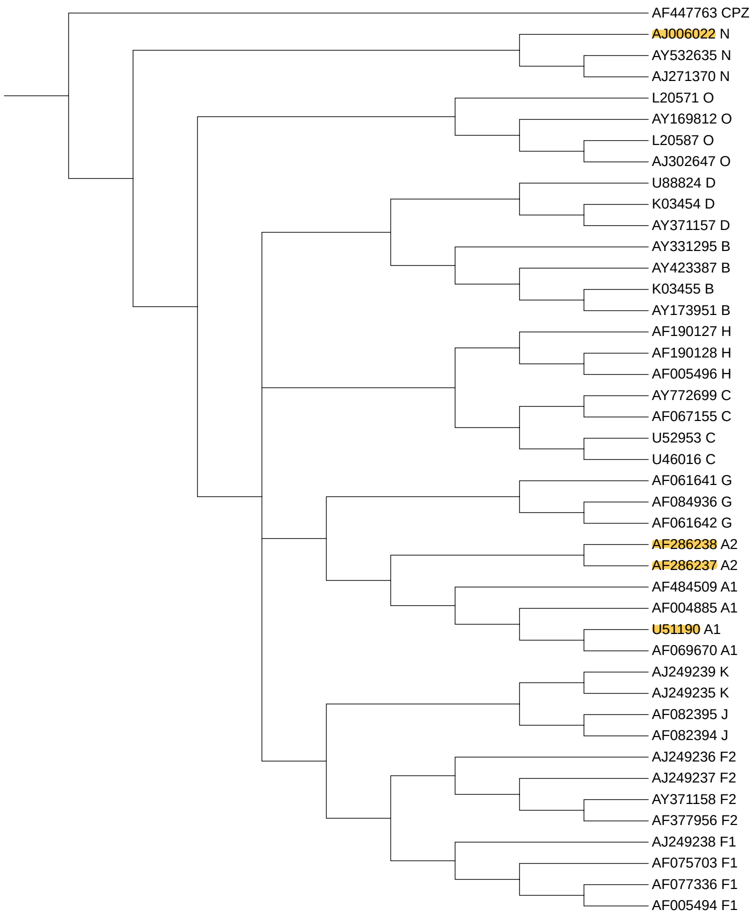


Figure 4 The phylogenetic tree on the 43 HIV sequences by PHYBWT [26, Figure 11]. Re-root the tree in CIV strain AF447763, as it is set outgroup in the reference tree in [45]. We highlight the strains that are used in the following experiments.

AJ006022 N of the HIV virus dataset, and Figure 7 considers the four taxa *BDBV 2012 KC545393, BDBV 2007 FJ217161, TAFV 1994 FJ217162, RESTV 1996 AB050936* of the Ebola virus dataset. For both datasets, we selected two taxa that are very similar (*AF286238 A2* and *AF286237 A2* for HIV, and *BDBV 2012 KC545393* and *BDBV 2007 FJ217161* for Ebola), one that is not too far from the first two (*U51190 A1* for HIV, and *TAFV 1994 FJ217162* for Ebola), and one last taxon that is far from every other considered taxon in the phylogeny (*AJ006022 N* for HIV, and *RESTV 1996 AB050936* for Ebola).

The algorithms for building McDag and computing the distributions were implemented in C++, compiled with `g++ 11.4.0` using the `-O3` and `-march=native` flags. The source code is available at `https://github.com/giovanni-buzzega/McDag` [12]. We carried out the experiments on a DELL PowerEdge R750 machine in a non-exclusive mode, featuring 24 cores with 2 Intel(R) Xeon(R) Gold 5318Y CPUs at 2.10 GHz, and 989 GB of RAM. The operating system is Ubuntu 22.04.2 LTS.

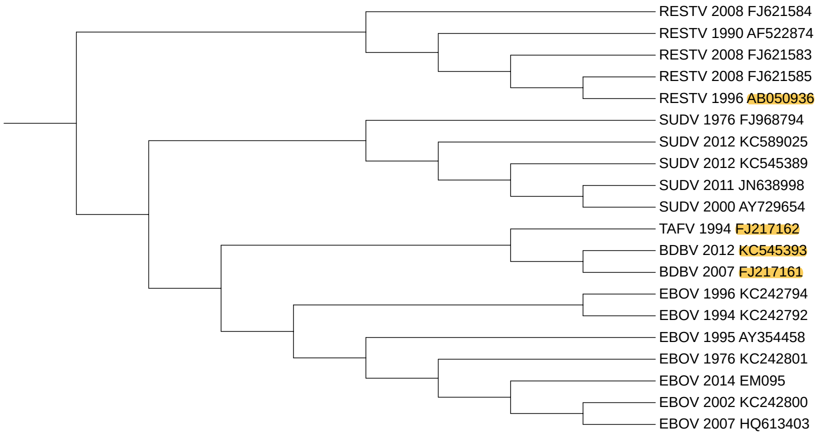


Figure 5 The phylogenetic tree on Ebolavirus dataset by PHYBWT [26, Figure 13]. We highlight the strains that are used in the following experiments.

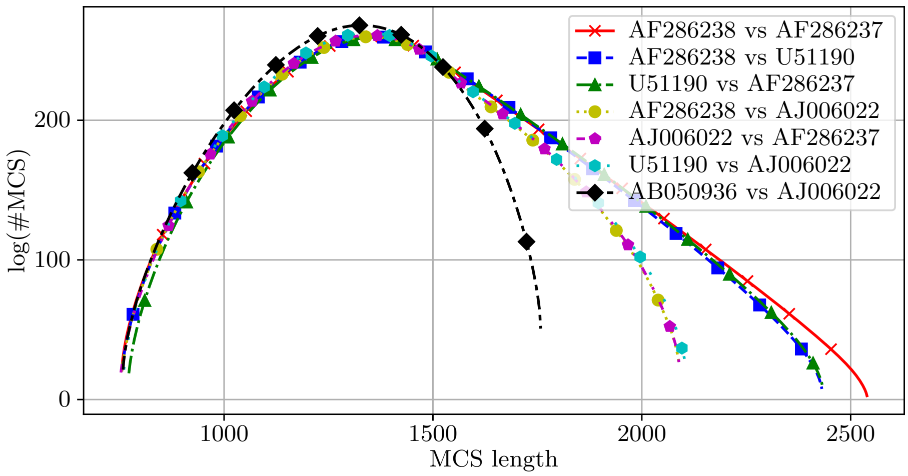


Figure 6 Length distribution of MCSs among the selected pairs of DNA sequences from the HIV dataset. The black line is the distribution of MCSs between HIV and Ebola virus taxa. The y-axis is logarithmic (base 10).

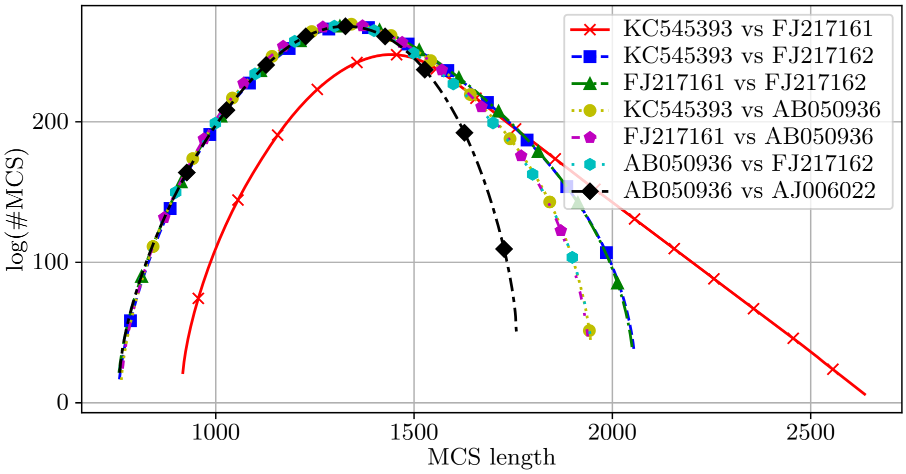


Figure 7 Length distribution of MCSs among different pairs of DNA sequences from the Ebola dataset. The black line is the distribution of MCSs between HIV and Ebola virus taxa. The y-axis is logarithmic (base 10).

For all strings we considered the substring between position 2500 and 5200 (chosen arbitrarily), and we built McDag. On average, index construction took 13.675 $\pm$ 0.673 seconds, followed by 15.763 $\pm$ 0.95 seconds to compute the MCS length distribution. As shown on the y-axis on both Figure 6 and Figure 7, the number of paths (and hence MCSs) in McDag is quite large, reaching values on the order of 10^{270}. Since such large numbers still remain within the representable range of a double, we did not use the the log-sum-exp trick in Section 4.2. However, when dealing with larger numbers of MCSs, we may have to resort to this technique to avoid overflow errors. In this case, the execution time may grow by a constant factor, due to the additional computational cost of the log and exp functions: in some preliminary testing on our data, we saw that the execution time increased to an average of 108.186 $\pm$ 10.973 seconds.

We now briefly discuss the outcome of the experiments. Since the LCS length is known to correlate well with string similarity [36], we see in both Figure 6 and 7, as expected, that the two strings considered most similar have the far right tail of their distribution ending at higher values on the x-axis. The most evident behaviour of the plots is that all lines, from left to right, start with a bell shape and, after the peak, decrease following a straight line before curving down again. This feature is more evident in the line that plots the distribution of MCSs between the taxa considered most similar.

Interestingly, we see that the two lines (in blue and green) that correspond to the MCS distribution between the taxon of medium distance and the first two taxa, are slightly detached from the first red line. For instance, in the case of HIV, Figure 6 shows an almost-perfect overlap up to lengths of 2100 on the x-axis; after that the lower similarity translates to a smaller number of long MCSs, with the straight part of the bell shape decaying earlier than in the red line. In the case of Ebola (Figure 7), there is again a gradual difference in where the lines drop on the right side (more similar pairs drop further right), but also a stark difference of the red line, representing the two closest taxa, in the left side of the graph: the start of the line is shifted right compared to the others, meaning that every MCS is longer than about 800. This behaviour suggests a particularly strong similarity, and further investigation into how it arises is an interesting direction of work.

The next three lines, in yellow, magenta, and cyan, represent the relations with the more dissimilar taxon. We see in both figures that the three lines again overlap, and the right side detaches from the other lines on lower values on the x-axis.

Finally, in both figures we added a black line that depicts the distribution of two completely unrelated string: in both plots, we used taxon *RESTV 1996 AB050936* of Ebola and taxon *AJ006022 N* of HIV. In both cases we have that the black line closely follows a bell shape, with no part of it showing a straight line behavior; moreover, on a large portion of the left side, it overlaps with the yellow, magenta and cyan lines.

This suggests that there is some baseline set of MCSs of any unrelated strings that acts as a *background noise*; after a given threshold length, the number of "non-noisy" MCSs seems to be a good indicator of string similarity, and, as an extension, of taxon similarity.

6 Conclusions

We have reviewed two recent results that use compact string indices to naturally highlight relevant information in a genomic context. The BWT-based approach PHYBWT infers evolutionary links by clustering similar substrings, and the DAG-based index MCDAG can be used to show the distribution of Maximal Common Subsequences, which exposes similarities among strings. We have also shown experimentally that MCS length distributions vary among closely related and more distantly related taxa, using the phylogeny generated by PHYBWT as a reference. Further work in visualizing and analysing the information emerging from these indices, as well as extending the analysis to new indices, is an interesting direction to explore and may yield positive results in phylogeny and, more generally, in the analysis of genomic sequences.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE, IEEE Computer Society, 2015. `doi:10.1109/FOCS.2015.14`.

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

3 Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pages 3–14. IEEE, 1995. `doi:10.1109/ICDE.1995.380415`.

4 H.-J. Bandelt and A. W. M. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in mathematics*, 92(1):47–105, 1992.

5 H.-J. Bandelt and A. W. M. Dress. Split decomposition: A new and useful approach to phylogenetic analysis of distance data. *Molecular Phylogenetics and Evolution*, 1(3):242–252, 1992. `doi:10.1016/1055-7903(92)90021-8`.

6 H.-J. Bandelt, K. T. Huber, J. H. Koolen, V. Moulton, and A. Spillner. *Basic Phylogenetic Combinatorics*. Cambridge University Press, 2012. URL: `http://www.cambridge.org/de/knowledge/isbn/item6439332/`.

7 M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. `doi:10.1016/j.tcs.2012.02.002`.

8 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000. doi:10.1109/SPIRE.2000.878178.

9 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 79–97. IEEE, 2015. doi:10.1109/FOCS.2015.15.

10 Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau. An FPT-algorithm for longest common subsequence parameterized by the maximum number of deletions. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.6.

11 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.

12 Giovanni Buzzega, Alessio Conte, Roberto Grossi, and Giulia Punzi. Mcdag: Indexing maximal common subsequences in practice. In *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, pages 21–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

13 Giovanni Buzzega, Alessio Conte, Roberto Grossi, and Giulia Punzi. McDag: indexing maximal common subsequences for k strings. *Algorithms for Molecular Biology*, 20(1):6, 2025. doi:10.1186/s13015-025-00271-z.

14 Giovanni Buzzega, Alessio Conte, Yasuaki Kobayashi, Kazuhiro Kurita, and Giulia Punzi. The complexity of maximal common subsequence enumeration. *Proc. ACM Manag. Data*, to appear, 2025.

15 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. Accepted to Festschrift's honoree Giovanni Manzini, 2025. arXiv:2506.01092.

16 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Polynomial-delay enumeration of maximal common subsequences. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 189–202, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32686-9_14.

17 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Enumeration of maximal common subsequences between two strings. *Algorithmica*, 84(3):757–783, 2022. doi:10.1007/s00453-021-00898-5.

18 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. A compact DAG for storing and searching maximal common subsequences. In Satoru Iwata and Naonori Kakimura, editors, *34th International Symposium on Algorithms and Computation, ISAAC 2023, December 3-6, 2023, Kyoto, Japan*, volume 283 of *LIPIcs*, pages 21:1–21:15. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ISAAC.2023.21.

19 Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph - overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003. doi:10.1016/S1570-8667(03)00029-7.

20 Maxime Crochemore and Zdeněk Troníček. Directed acyclic subsequence graph for multiple texts. *Rapport IGM*, pages 99–13, 1999.

21 L. Egidi, F. A. Louza, G. Manzini, and G. P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019. doi:10.1186/s13015-019-0140-0.

22 Campbell Fraser, Robert W. Irving, and Martin Middendorf. Maximal common subsequences and minimal common supersequences. *Inf. Comput.*, 124(2):145–153, 1996. doi:10.1006/inco.1996.0011.

23 Stephen K. Gire, Augustine Goba, Kristian G. Andersen, Rachel S. G. Sealfon, Daniel J. Park, Lansana Kanneh, Simbirie Jalloh, Mambu Momoh, Mohamed Fullah, Gytis Dudas, Shirlee Wohl, Lina M. Moses, Nathan L. Yozwiak, Sarah Winnicki, Christian B. Matranga, Christine M. Malboeuf, James Qu, Adrianne D. Gladden, Stephen F. Schaffner, Xiao Yang, Pan-Pan Jiang, Mahan Nekoui, Andres Colubri, Moinya Ruth Coomber, Mbalu Fonnie, Alex Moigboi, Michael Gbakie, Fatima K. Kamara, Veronica Tucker, Edwin Konuwa, Sidiki Saffa, Josephine Sellu, Abdul Azziz Jalloh, Alice Kovoma, James Koninga, Ibrahim Mustapha, Kandeh Kargbo, Momoh Foday, Mohamed Yillah, Franklyn Kanneh, Willie Robert, James L. B. Massally, Sinéad B. Chapman, James Bochicchio, Cheryl Murphy, Chad Nusbaum, Sarah Young, Bruce W. Birren, Donald S. Grant, John S. Scheiffelin, Eric S. Lander, Christian Happi, Sahr M. Gevao, Andreas Gnirke, Andrew Rambaut, Robert F. Garry, S. Humarr Khan, and Pardis C. Sabeti. Genomic surveillance elucidates ebola virus origin and transmission during the 2014 outbreak. *Science*, 345(6202):1369–1372, 2014. doi:10.1126/science.1259657.

24 Ronald I. Greenberg. Bounds on the number of longest common subsequences. *CoRR*, cs.DM/0301030, 2003. arXiv:cs/0301030.

25 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT: Alignment-Free Phylogeny via eBWT Positional Clustering. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *LIPIcs*, pages 23:1–23:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2022.23.

26 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. doi:10.1186/S13015-023-00232-4.

27 Miyuji Hirota and Yoshifumi Sakai. Efficient algorithms for enumerating maximal common subsequences of two strings. *CoRR*, abs/2307.10552, 2023. doi:10.48550/arXiv.2307.10552.

28 Miyuji Hirota and Yoshifumi Sakai. A fast algorithm for finding a maximal common subsequence of multiple strings. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 106(9):1191–1194, 2023. doi:10.1587/transfun.2022dml0002.

29 D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, October 2005.

30 Michelle Kendall and Caroline Colijn. Mapping Phylogenetic Trees to Reveal Distinct Patterns of Evolution. *Molecular Biology and Evolution*, 33(10):2735–2743, 2016. doi:10.1093/molbev/msw124.

31 Thomas Leitner, Bette Korber, Marcus Daniels, Charles Calef, and Brian Foley. HIV-1 Subtype and Circulating Recombinant Form (CRF) Reference Sequences, 2005. URL: https://www.hiv.lanl.gov/content/sequence/HIV/REVIEWS/LEITNER2005/leitner.html.

32 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.

33 David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978. doi:10.1145/322063.322075.

34 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *ACM-SIAM SODA*, pages 319–327, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

35 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theoret. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

36 Mike Paterson and Vlado Dančík. Longest common subsequences. In *International symposium on mathematical foundations of computer science*, pages 127–142. Springer, 1994.

37 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler transform. *BMC Bioinformatics*, 21, 2020. doi:10.1186/s12859-020-03586-3.

38 Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992. doi:10.1016/0304-3975(92)90142-3.

39 Yoshifumi Sakai. Maximal Common Subsequence Algorithms. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CPM.2018.1`.

40 Yoshifumi Sakai. Maximal common subsequence algorithms. *Theor. Comput. Sci.*, 793:132–139, 2019. `doi:10.1016/j.tcs.2019.06.020`.

41 Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

42 Zdenek Tronícek. Common subsequence automaton. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata, 7th International Conference, CIAA 2002, Tours, France, July 3-5, 2002, Revised Papers*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275. Springer, Springer, 2002. `doi:10.1007/3-540-44977-9_28`.

43 Tandy Warnow. *Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation*. Cambridge University Press, 2017.

44 R. Wittler. Alignment- and Reference-Free Phylogenomics with Colored de Bruijn Graphs. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143, pages 2:1–2:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.WABI.2019.2`.

45 Xiaomeng Wu, Zhipeng Cai, Xiu-Feng Wan, Tin Hoang, Randy Goebel, and Guohui Lin. Nucleotide composition string selection in HIV-1 subtyping using whole genomes. *Bioinformatics*, 23(14):1744–1752, May 2007. `doi:10.1093/bioinformatics/btm248`.

46 Andrzej Zielezinski, Hani Z Girgis, Guillaume Bernard, Chris-Andre Leimeister, Kujin Tang, Thomas Dencker, Anna Katharina Lau, Sophie Röhling, Jae Jin Choi, Michael S Waterman, Matteo Comin, Sung-Hou Kim, Susana Vinga, Jonas S Almeida, Cheong Xin Chan, Benjamin T James, Fengzhu Sun, Burkhard Morgenstern, and Wojciech M Karlowski. Benchmarking of alignment-free sequence comparison methods. *Genome Biology*, 20:144, 2019. `doi:10.1186/s13059-019-1755-7`.