


# phyBWT: Alignment-Free Phylogeny via eBWT Positional Clustering

Veronica Guerrini<sup>1</sup> ✉ 

Dipartimento di Informatica, University of Pisa, Italy

Alessio Conte ✉ 

Dipartimento di Informatica, University of Pisa, Italy

Roberto Grossi ✉ 

Dipartimento di Informatica, University of Pisa, Italy

Gianni Liti ✉ 

CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, France

Giovanna Rosone<sup>1</sup> ✉ 

Dipartimento di Informatica, University of Pisa, Italy

Lorenzo Tattini ✉ 

CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, France

---

## Abstract

Molecular phylogenetics is a fundamental branch of biology. It studies the evolutionary relationships among the individuals of a population through their biological sequences, and may provide insights about the origin and the evolution of viral diseases, or highlight complex evolutionary trajectories.

In this paper we develop a method called **phyBWT**, describing how to use the extended Burrows-Wheeler Transform (eBWT) for a collection of DNA sequences to directly reconstruct phylogeny, bypassing the alignment against a reference genome or de novo assembly. Our **phyBWT** hinges on the combinatorial properties of the eBWT positional clustering framework. We employ eBWT to detect relevant blocks of the longest shared substrings of varying length (unlike the  $k$ -mer-based approaches that need to fix the length  $k$  a priori), and build a suitable decomposition leading to a phylogenetic tree, step by step. As a result, **phyBWT** is a new alignment-, assembly-, and reference-free method that builds a partition tree without relying on the pairwise comparison of sequences, thus avoiding to use a distance matrix to infer phylogeny.

The preliminary experimental results on sequencing data show that our method can handle datasets of different types (short reads, contigs, or entire genomes), producing trees of quality comparable to that found in the benchmark phylogeny.

**2012 ACM Subject Classification** Applied computing → Bioinformatics; Mathematics of computing → Combinatorial algorithms

**Keywords and phrases** Phylogeny, partition tree, BWT, positional cluster, alignment-free, reference-free, assembly-free

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2022.23

**Supplementary Material** *Software (Source Code)*: <https://github.com/veronicaguerrini/phyBWT>

**Funding** Work partially supported by Italian Ministry for Education and Research – PRIN Project n. 20174LF3T8 AHeAD and by University of Pisa - Project no. PRA\_2020–2021\_26 “Metodi Informatici Integrati per la Biomedica”.

---

<sup>1</sup> Corresponding author



## 1 Introduction

Molecular phylogenetics are a key tool for understanding the evolutionary relationships among biological sequences. Phylogenies, in the form of rooted or unrooted trees, can be used for several purposes: to reconstruct the ancestry of the species (or other taxa) on the tree of life, to understand the epidemiological dynamics of pathogens, and to identify and study complex evolutionary events such as hybridisation [17, 34], introgression [12], and horizontal gene transfer [33]. Thus, they are successfully employed in almost every branch of biology, including e.g. population genomics and metagenomics, ecology, and biogeography [40].

A vast array of techniques for inferring phylogeny have been developed over the years [37]. Sequence-based methods analyze the DNA or RNA sequences of the taxa, and are based on their similarity or dissimilarity detection. Most of them rely on a distance matrix by computing the pairwise evolutionary distances between every pair of input sequences. Standard algorithms, such as the neighbour-joining algorithm [32], are then applied to the distance matrix to perform the tree reconstruction.

A crucial component is how to compute these evolutionary distances. Sequence alignment is a central task in distance computation, which is performed on either entire sequences or parts of them, with the optional usage of a reference genome. With the advent of new sequencing technologies and the completion of various genome projects, the number of whole-genome sequence data available has increased and a new era for phylogeny started. Owing to the rising cost of the alignment task, alignment-free approaches for quantifying the similarity/dissimilarity between sequences have been introduced: an advantage of these approaches is that they are robust for recombination and shuffling events [36, 35, 42]. As the majority of alignment-free approaches for phylogenetic reconstruction transforms each sequence into a multiset of  $k$ -mers, *i.e.* substrings of length  $k$  extracted from the input sequences, they can also work directly on the reads obtained from the sequencing platforms, without the need of performing a preliminary assembly of these reads.

In this paper we introduce **phyBWT**, which combines many features in a single new method to reconstruct a phylogenetic tree. Firstly it is alignment-, assembly-, and reference-free. Second, it does not need a distance matrix as it does not rely on the pairwise comparison of sequences. Third, it exploits the combinatorial properties of the *positional clustering* framework recently introduced in [28], overcoming the limitations of employing  $k$ -mers with fixed size  $k$  a priori.

The contribution of this paper is twofold, theoretical as well as practical. To the best of our knowledge, **phyBWT** is the first method that applies the properties of the Extended Burrows-Wheeler Transform (eBWT), employed in the positional clustering, to the idea of decomposition for phylogenetic inference. Moreover, **phyBWT** not only is oblivious to extra information such as reference sequences or read mappings, but it also avoids any assembly or alignment of input sequences. Finally, **phyBWT** infers the tree structure by comparing all the sequences simultaneously and efficiently, instead of performing their pairwise comparisons. We present a preliminary experimental evaluation of **phyBWT** for reconstructing phylogenetic trees using different types of biological sequences (short reads and de novo assemblies/genomes) and different taxa. The phylogenetic trees produced by **phyBWT** are of high quality according to the benchmarks in the literature.

### State of the art

The Burrows-Wheeler Transform (BWT) [9] of a string (and the eBWT of a set of strings [25, 5]) is a suitable permutation of the symbols of the string(s), whose output shows a local similarity, *i.e.* symbols preceding similar contexts tend to occur in clusters. Both transformations

have been intensively studied with important and successful applications in several areas. For instance, the eBWT has been used for defining alignment-free methods based on a pairwise distance matrix [25, 26, 39, 19] in order to build up a phylogenetic tree for mitochondrial DNA genomes. The positional clustering detects “interesting” blocks in the output of the eBWT [25, 5], so that the requirement on the fixed size  $k$  in  $k$ -mers is relaxed and becomes of variable-order, not fixed a priori, in an adaptive way according to the contexts. This framework has already been used in other bioinformatics tasks, such as for detecting SNPs and INDELS in short-read datasets [29] and for lossy compression of FASTQ datasets [18].

We observe that phyBWT exploits the underlying properties of the eBWT: (i) the clustering effect, *i.e.*, the fact that the eBWT tends to group together equal symbols in the transformed string that occur in similar contexts in the input string collection; (ii) the fact that if a substring  $x$  occurs in one or more strings, then the suffixes of the input dataset starting with  $x$ -occurrence are close in the sorted list of suffixes. In other words, the greater the number of these substrings shared by two taxa is, the more they are similar.

Although phyBWT does not use the distance matrix, it has some resemblance with split decomposition methods when reconstructing the tree from the information gathered through the eBWT. We recall that split decomposition relies on a solid mathematical ground [2, 4], and has been successfully applied to phylogeny [3]. The idea is to score the possible splits (*i.e.* bipartitions) of the taxa, and assign an isolation index to each split based on the distances in the given matrix. Compatible splits are those with an empty intersection on one of the parts in the splits, and the isolation index is treated as a priority weight in making a (greedy) choice among the splits. Compatible splits induce a tree and vice versa. However, a residual error is generated on real-world data, and a notion of weak split compatibility is preferred to create a weighted phylogeny network instead of a phylogeny tree: the shortest weighted part between any two nodes in this network gives the isolation index in the corresponding split. For  $\ell$  taxa, only  $O(\ell^2)$  splits are needed for split decomposition instead of  $2^\ell$  ones [2].

As the original algorithm in the seminal papers on split decomposition [2, 3] requires  $O(\ell^6)$  comparisons, further papers have addressed efficiency and extended these ideas. The recent alignment-free method SANS [38, 31] uses the notions of the split decomposition theory to greedily build a list of weakly compatible splits from which to infer phylogenies. In the list, each split has its own weight computed by counting  $k$ -mers that are stored in a colored de Bruijn graph [38] (this has been improved later by hashing [31], leaving the colored de Bruijn graph as input option). The calculated list of splits ordered by weight is then filtered according to two strategies that are described and implemented in the software tool SplitsTree [20]. In our experimental study, we compare the trees obtained by SANS and phyBWT. It should be noted that SANS is also able to reconstruct phylogenetic networks whereas phyBWT focuses on phylogenetic trees only.

As previously mentioned, a plethora of methods have been designed for phylogeny (*e.g.* DBLP reports over 500 papers having “phylogeny” in the title). We refer the reader to [21, 22, 37] for a complete and detailed review of various methods for phylogeny estimation. We briefly mention here that among the alignment-based approaches are character-based methods [40], that generally produce alignments of the input sequences and compare all sequences simultaneously considering one character per time (*e.g.* using maximum parsimony or maximum likelihood). The alignment-free tree reconstruction comes from computing some distribution within and among  $k$ -mers by using a distance matrix or not. For instance, the method in [14] reconstructs a phylogeny from whole-genome short-read sequencing data on the basis on a matrix of pairwise genetic distances, without assembling the reads.

## 2 Notation and background

### 2.1 Notation

Let  $s$  be a string (also called sequence) of length  $n$  on the alphabet  $\Sigma$ . We denote the  $i$ -th symbol of  $s$  by  $s[i]$ . A *substring* of any  $s$  is denoted as  $s[i, j] = s[i] \cdots s[j]$ , with  $s[1, j]$  being called a *prefix* and  $s[i, n + 1]$  a *suffix* of  $s$ . A  $k$ -mer is a string of length  $k$ .

Let  $\mathcal{S} = \{s_1, s_2, \dots, s_\ell\}$  be a collection of  $\ell$  strings. We assume that each string  $s_i \in \mathcal{S}$  has length  $n_i$  and is followed by a special end-marker symbol  $S_i[n_i + 1] = \$_i$ , which is lexicographically smaller than any other symbol in  $\mathcal{S}$ , and does not appear in  $\mathcal{S}$  elsewhere<sup>2</sup>.

### 2.2 Basic data structures

The *Burrows-Wheeler Transform* (BWT) [9] is a well-known widely used reversible string transformation that can be extended to a collection of strings. Such an extension, introduced in [25], is a reversible transformation whose output string (denoted by  $\text{ebwt}(\mathcal{S})$ ) is a permutation of the symbols of all strings in  $\mathcal{S}$ . In [5], the authors introduced a variant of this transformation for string collection in which a distinct end-marker is appended to each string, making the collection ordered. Such transformations are known as eBWT or multi-string BWT.

The length of  $\text{ebwt}(\mathcal{S})$  is denoted by  $N = \sum_{i=1}^{\ell} (n_i + 1)$ , and  $\text{ebwt}(\mathcal{S})[i] = x$ , with  $1 \leq i \leq N$ , if  $x$  circularly precedes the  $i$ -th suffix  $S_j[k, n_j + 1]$  (for some  $1 \leq j \leq \ell$  and  $1 \leq k \leq n_j + 1$ ), according to the lexicographic sorting of the suffixes of all strings in  $\mathcal{S}$ .

Usually the output string  $\text{ebwt}(\mathcal{S})$  is enhanced with the *document array* (DA) and *longest common prefix* (LCP) array of  $\mathcal{S}$ .

The *document array* of  $\mathcal{S}$  (denoted by  $\text{da}(\mathcal{S})$ ) is the array of length  $N$  such that  $\text{da}(\mathcal{S})[i] = j$ , with  $1 \leq j \leq \ell$  and  $1 \leq i \leq N$ , where  $\text{ebwt}(\mathcal{S})[i]$  is a symbol of the string  $s_j$ .

The *longest common prefix* (LCP) array [24] of  $\mathcal{S}$  is the array  $\text{lcp}(\mathcal{S})$  of length  $N + 1$ , such that  $\text{lcp}(\mathcal{S})[i]$ , with  $2 \leq i \leq N$ , is the length of the longest common prefix between the suffixes associated with the positions  $i$  and  $i - 1$  in  $\text{ebwt}(\mathcal{S})$ , and  $\text{lcp}(\mathcal{S})[1] = \text{lcp}(\mathcal{S})[N + 1] = 0$  by default. The set  $\mathcal{S}$  can be omitted when it is clear from the context.

The following is an important property of the BWT, and thus of the related data structures DA and LCP, that will be used in our method:

► **Remark 1.** One can obtain the DA of a subset of  $\mathcal{S}$  by scanning the  $\text{DA}(\mathcal{S})$ . In [5], the authors prove that given a collection  $\mathcal{S} = \{S_1, S_2, \dots, S_\ell\}$  of strings and  $\text{ebwt}(\mathcal{S})$ , one can obtain the eBWT of a subset  $\mathcal{R}$  of  $\mathcal{S}$  by removing all the characters not in  $\mathcal{R}$ , without constructing the eBWT from scratch, as the relative order of suffixes holds. Similarly, one can obtain the LCP of a subset of  $\mathcal{S}$  by using the properties of the LCP array.

Let  $\mathcal{R} \subset \mathcal{S}$ . We denote by  $\text{ebwt}(\mathcal{S})|_{\mathcal{R}}$  (resp.  $\text{da}(\mathcal{S})|_{\mathcal{R}}$ ,  $\text{lcp}(\mathcal{S})|_{\mathcal{R}}$ ) the restriction of the data structure  $\text{ebwt}(\mathcal{S})$  (resp.  $\text{da}(\mathcal{S})$ ,  $\text{lcp}(\mathcal{S})$ ) to the set of strings  $\mathcal{R}$ .

### 2.3 LCP-interval and k-mer vs Positional cluster

The LCP-intervals [1] of lcp-value  $k$ , or  $k$ -intervals, are maximal intervals  $[i, j]$  that satisfy  $\text{lcp}(\mathcal{S})[r] \geq k$  for  $i < r \leq j$ . In other words, the suffixes associated with  $k$ -intervals have a common  $k$ -mer as prefix.

<sup>2</sup> Note that, in the implementations, one can use a single symbol as end-marker for all strings, but end-markers from different strings are then sorted on the basis of the index and the relative order of the strings in the set they belong to.

In any string collection, thus, LCP-intervals of lcp-value  $k$  are in a one-to-one correspondence with the set of all  $k$ -mers.

Note that the common prefix  $w$  in a LCP-interval is of length at least  $k$ , but it could be longer. So, to overcome the limitation of strategies based on LCP-intervals that require to fix the length  $k$ , the authors of [28, 29] introduced a new framework called “positional clustering”. In this framework the intervals do not depend on a value  $k$  fixed a-priori, but they are enclosed between two “local minima” in the LCP-array (thus, their boundaries are data-driven).

Crucially, the length  $k$  of the common prefix  $w$  of the suffixes inside such intervals is not the same, but it differs interval by interval. Hence, there is no one-to-one correspondence between such intervals and the set of  $k$ -mers.

However, as to exclude intervals corresponding to some short random contexts  $w$ , one needs to set a minimum length for  $w$ , which we denote by  $k_m$ .

According to [29], an *eBWT positional cluster*  $\text{eBWTCLUST}[i, j]$  is a maximal substring  $\text{ebwt}[i, j]$  where  $\text{lcp}[r] \geq k_m$ , for all  $i < r \leq j$ , and none of the indices  $r$ ,  $i < r \leq j$ , is a *local minimum* of the LCP array.

By definition, we have that:

► **Remark 2.** Any two  $\text{eBWT}$  positional clusters,  $\text{eBWTCLUST}[i, j]$  and  $\text{eBWTCLUST}[i', j']$ , such that  $i \neq i'$  are disjoint, *i.e.* it holds that either  $j < i'$  or  $j' < i$ .

Here, we define a local minimum of the LCP array (of length  $N$ ) any index  $i$ ,  $1 < i < N$  such that  $\text{lcp}[i - 1] > \text{lcp}[i]$  and  $\text{lcp}[i] < \text{lcp}[i + j]$ , where  $j > 1$  is the number of adjacent occurrences of the value  $\text{lcp}[i]$  from position  $i$ . For instance, let  $\text{lcp} = [2, 1, 3, 3, 5, 4, 2, 2, 7]$ . The local minima are indices 2 and 7.

Note that the above definition differs from that in [29], where local minima in the LCP array (of length  $N$ ) are detected searching for indices  $r$  such that  $\text{lcp}[r - 1] \geq \text{lcp}[r] < \text{lcp}[r + 1]$ , for all  $1 < r \leq N$ . According to such definition, local minima can be detected in any non-increasing sequence where some values are repeated. For instance, for the second occurrence of 4 in the sequence 5, 4, 4, 2 yields the definition of local minimum. Therefore, the slightly different notion of local minima we use is to maximize the length of the non-increasing sequence described in the following Remark 3.

► **Remark 3** ([28], Thm 3.3). In any  $\text{eBWT}$  positional cluster, the lcp-values form a sequence of non-decreasing values followed by a (possibly empty) sequence of non-increasing values.

From the above remark follows that the length  $l$  of the longest common prefix shared by the suffixes associated with a  $\text{eBWT}$  positional cluster  $\text{ebwt}[i, j]$  is given by the minimum value in  $\text{lcp}[i + 1, j]$ , which could be simply obtained by taking the minimum between the values  $\text{lcp}[i + 1]$  and  $\text{lcp}[j]$ .

In general, if we set the minimum length  $k_m$  equal to  $k$ , the set of  $\text{eBWT}$  positional clusters forms a refinement of the set of  $\text{ebwt}[i, j]$  with  $[i, j]$  LCP-interval of lcp-value  $k$ .

In fact, any  $\text{ebwt}[i, j]$ , where  $[i, j]$  is a LCP-interval, can be subdivided in correspondence of the local minima of  $\text{lcp}[i, j]$ , thus giving rise to a sequence of consecutive  $\text{eBWT}$  positional clusters (see Figure 1). Clearly, such subdivision depends only on the trend of the LCP values inside the LCP-interval  $[i, j]$ . Hence, more than one positional cluster can be related to the same LCP-interval, and equivalently, to the same  $k$ -mer.

► **Example 4** (running example). In Figure 1, we represent the data structures used in our tool (`cda`, `ebwt`, `lcp`), the auxiliary array `da` and the sorted list of suffixes, for the sake of clarity. The LCP-intervals of lcp-value  $k = 1$  correspond to the following intervals:  $[4, 10]$ ,  $[11, 17]$ ,  $[18, 28]$ ,  $[29, 34]$ . Whereas the horizontal lines delimit  $\text{eBWTCLUST}$  for  $k_{min} = 1$ .

$i$	cda	da	lcp	ebwt	Sorted suffixes	$i$	cda	da	lcp	ebwt	Sorted suffixes
1	1	1	0	A	\$	18	3	3	0	C	GACT\$
2	2	2	0	T	\$	19	3	3	2	C	GAGTACGACT\$
3	3	3	0	T	\$	20	1	1	1	G	GCGTACCA\$
4	1	1	0	C	A\$	21	2	2	5	G	GCGTATT\$
5	1	1	1	T	ACCA\$	22	1	1	1	\$	GGCGTACCA\$
6	3	3	2	T	ACGACT\$	23	2	2	6	G	GCGTATT\$
7	3	3	4	\$	ACGAGTACGACT\$	24	2	2	2	G	GGGCGTATT\$
8	3	3	2	G	ACT\$	25	2	2	3	\$	GGGCGTATT\$
9	3	3	1	G	AGTACGACT\$	26	1	1	1	C	GTACCA\$
10	2	2	1	T	ATT\$	27	3	3	4	A	GTACGACT\$
11	1	1	0	C	CA\$	28	2	2	3	C	GTATT\$
12	1	1	1	A	CCA\$	29	2	2	0	T	T\$
13	3	3	1	A	CGACT\$	30	3	3	1	C	T\$
14	3	3	3	A	CGAGTACGACT\$	31	1	1	1	G	TACCA\$
15	1	1	2	G	CGTACCA\$	32	3	3	3	G	TACGACT\$
16	2	2	4	G	CGTATT\$	33	2	2	2	G	TATT\$
17	3	3	1	A	CT\$	34	2	2	1	A	TT\$

■ **Figure 1** Extended Burrows-Wheeler Transform (EBWT), LCP array, and the auxiliary data structures DA and CDA for the set  $\mathcal{S} = \{\text{GGCGTACCA}, \text{ACGAGTACGACT}, \text{GGGCGTATT}\}$ .

Note that when  $k_{min} = k$ , the eBWTCLUST can refine the LCP-intervals. For example the LCP-interval  $[18, 28]$  includes five positional clusters: eBWTclust $[18, 19]$ , eBWTclust $[20, 21]$ , eBWTclust $[22, 23]$ , eBWTclust $[24, 25]$ , eBWTclust $[26, 28]$ .

### 3 Method

In this section, we describe the proposed method for building a phylogenetic tree where each leaf, representing an organism, is a set of strings (*e.g.* sequencing reads, contigs, genome).

The idea of our method is to reconstruct the tree through a series of partitions performed on groups of organisms. As these partitions isolate groups of organisms from each other, we proceed in both directions: we divide each part in one direction, and we group the parts together in the other direction. Each part corresponds to a node of the phylogeny tree. When it is not possible to further divide or group together, we draw the edges of the tree from those groups to a node corresponding to their union.

The partitions generated by our method are intended to estimate phylogenetic signals; in particular, each part produces evidence of separations among the input set of nodes.

More formally, we denote the set of leaves as  $\mathcal{S} = \{S_1, S_2, \dots, S_\ell\}$ , where each leaf  $S_i \in \mathcal{S}$  contains  $m_i$  strings including their reverse-complement, *i.e.*  $S_i = \{s_{i,1}, \dots, s_{i,m_i}\}$ , where  $s_{i, \frac{m_i}{2}}, \dots, s_{i,m_i}$  are strings in the reverse-complement form. Each node of the tree identifies a set of organisms (*i.e.* it is a subset of  $\mathcal{S}$ ).

Subsection 3.1 describes how the partitioning procedure is iteratively called to infer a phylogeny tree for  $\mathcal{S}$ , while Subsection 3.2 provides the details of the inner partition algorithm.

The idea behind the partitioning algorithm, described in Subsection 3.2, is to group together nodes whose associated strings share long common substrings of varying length which are *not* present in other nodes, and we interpret the presence of such substrings as a common feature of the group that differentiates it from the others.

To perform partitions we do not use any external information, such as reference sequences or annotations, and in addition, we do not perform assembly or alignment.

Finally, we recall that, in any tree, a *cut* of a given subset of its edges determines a partition of the set of its leaves. Therefore, the notion of partition of  $\mathcal{S}$  we use slightly differs from the one of split, as a split of the set of leaves is obtained by removing a single edge from the corresponding tree. Thus, splits are essentially bi-partitions of  $\mathcal{S}$ .



■ **Algorithm 1** TreeReconstruction( $\mathcal{S}$ ).

---

```

input :  $\mathcal{S} = \{S_1, \dots, S_\ell\}$ 
output : A tree whose leaves are the elements of  $\mathcal{S}$ 

1 Initialize  $\mathcal{Q} \leftarrow \{\{S_1\}, \dots, \{S_\ell\}\}$ 
2 Queue.push( $\mathcal{Q}$ )
3 while Queue is not empty do
4    $\{Q_1, \dots, Q_q\} \leftarrow$  Queue.pop()
5   for  $i \in \{1, \dots, q\}$  do Create node  $Q_i$  if it does not exists
6   if  $q > 1$  then
7      $\mathcal{P} \leftarrow$  PARTITION( $\{Q_1, \dots, Q_q\}$ )
8      $p \leftarrow \mathcal{P}.size()$ 
9     if  $p = q$  then /* the partitioning cannot aggregate further */
10      Create node  $\mathcal{Q} = \bigcup\{Q_1, \dots, Q_q\}$  if it does not exists
11      Draw edges from the node  $\mathcal{Q}$  to nodes  $Q_i$ , for all  $i \in \{1, \dots, q\}$ 
12    else
13      Queue.push( $\{\bigcup P_1, \dots, \bigcup P_p\}$ ) /* link partitions to each other */
14      for  $i \in \{1, \dots, p\}$  do
15        Queue.push( $P_i$ ) /* link elements within the partition */

```

---

### 3.1 Partitioning-based tree reconstruction

We here describe a method that, given a partitioning algorithm for sets of strings as a blackbox, is able to reconstruct a tree by applying the partitioning step by step.

The blackbox for partitioning is described in Subsection 3.2, and denoted here by PARTITION. Given a set  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , the output of PARTITION is a non-empty collection  $\mathcal{P}$  of subsets of  $\mathcal{Q}$ , *i.e.*  $\mathcal{P} = \{P_1, \dots, P_p\}$ , such that

1.  $\bigcup_{i=1}^p P_i = \mathcal{Q}$ , *i.e.*, the union of all  $P_i \in \mathcal{P}$  is  $\mathcal{Q}$ .
2.  $P_i \cap P_j = \emptyset$  for all  $i \neq j$ , *i.e.*, every pair of sets in  $\mathcal{P}$  has empty intersection.

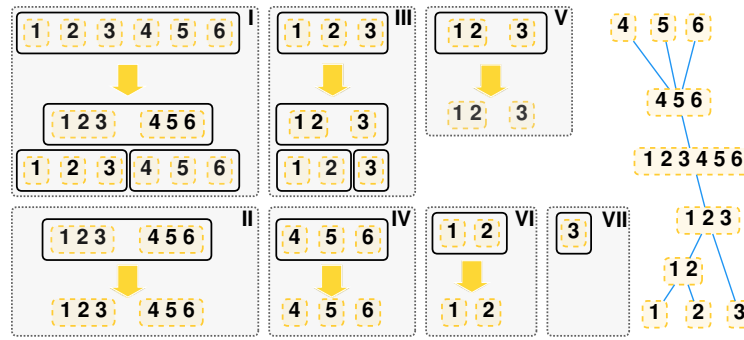
The input set  $\mathcal{Q}$  is a collection of sets of organisms. The key in this procedure is that each part  $P_i$  of  $\mathcal{Q}$  generated by PARTITION groups sets of organisms that share similarity to each other and divergence from the others. As the trivial partition  $\mathcal{P} = \{\mathcal{Q}\}$  provides no significant information, we assume PARTITION is such that each part  $P_i$  will be a proper subset of  $\mathcal{Q}$ .

Formally, each element  $Q_i$  will identify a node of the tree, and thus, a set of organisms (*i.e.* a subset of  $\mathcal{S}$ ). In particular, each leaf  $Q_i = \{S_i\}$ , will be a singleton set containing a single organism, while each internal node  $Q_i$  will be the union of the leaves of its subtree.

For the sake of clarity, we explicitly create the nodes of the tree (by their corresponding set of organisms) while reconstructing it (see Algorithm 1).

For convenience, given a collection of sets  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , we will use the notation  $\bigcup \mathcal{Q}$  as a shorthand for  $Q_1 \cup \dots \cup Q_q$ , *i.e.*, the set of all elements of the sets of  $\mathcal{Q}$ .

**Algorithm structure.** To give an intuitive overview, the algorithm iteratively considers several sets of organisms  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$  and aggregates them into *groups of sets* of organisms by means of PARTITION, obtaining a partition  $\mathcal{P} = \{P_1, \dots, P_p\}$  with each  $P_i \subset \mathcal{Q}$ . At each call of PARTITION, if the output  $\mathcal{P}$  is not the finest partition on  $\mathcal{Q}$  (*i.e.*  $p < q$ ), then it proceeds in two directions:



■ **Figure 2** A possible execution of Algorithm 1 on a set of organisms  $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ . Each panel shows an iteration of the algorithm: the rectangle at the top is the element taken from the queue, and the arrows points to the elements generated from the queue; dashed sets within the queue elements represent the sets of organisms. Iterations II, IV, V and VI meet the condition in Line 9: no new element of the queue is produced (we only show the result of PARTITION) and edges are generated. To the right is the resulting tree.

- Each part  $P_i$  will be a node of the tree (identified by the union of its contained  $Q_j$  sets), and the set  $\mathcal{P}$  is recursively fed to PARTITION.
- Within each single part  $P_i$ , each set  $Q_j \in P_i$  is a node, and these nodes will be recursively aggregated by calling PARTITION on  $P_i$  to induce a tree structure.

Otherwise, if the output  $\mathcal{P}$  is the partition of singletons (*i.e.* each  $P_i$  is a singleton), that means PARTITION is not able to further aggregate the elements of  $\mathcal{Q}$  (for instance, when called on 1 or 2 sets of organisms), then it traces edges between the node  $\mathcal{Q}$  and those corresponding to the sets  $Q_i$ .

At the very beginning, the set  $\mathcal{Q}$  is made of all singletons corresponding to the leaves, *i.e.*, the single organisms in  $\mathcal{S}$ , so  $\mathcal{Q} = \{\{S_1\}, \dots, \{S_\ell\}\}$ . The set  $\mathcal{Q}$  is inserted into a queue. The algorithm will iteratively process elements of the queue and will generate new elements for the queue, as described above (*i.e.*, one with the set  $\mathcal{P}$  of parts, and one for each  $P_i$ ).

The pseudocode in Algorithm 1 describes the method in detail. An example run is provided in Figure 2.

### 3.2 Inner Partitioning algorithm

In this subsection, we describe the approach we use as inner function within the algorithm described in Subsection 3.1 in order to obtain a partition  $\mathcal{P} = \{P_1, \dots, P_p\}$  of disjoint subsets from  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , where  $q \geq p > 1$ .

According to Subsection 3.1, the set  $\mathcal{Q}$  to be partitioned comprises some (not necessarily all) organisms, which are distributed among  $Q_1, \dots, Q_q$ .

Before describing the inner partitioning procedure, we introduce some notation.

We recall that the set  $\mathcal{S}$  contains  $\ell$  sets of strings, as for each organism we can have multiple strings (like reads, contigs, a genome, and so on). So,  $\mathcal{S} = \{S_1, S_2, \dots, S_\ell\}$  and each set  $S_i \in \mathcal{S}$  contains  $m_i$  strings, *i.e.*  $S_i = \{s_{i,1}, \dots, s_{i,m_i}\}$ . Note that the definitions of eBWT, LCP and DA apply also to this case, *e.g.*  $\text{ebwt}(\mathcal{S}) = \text{ebwt}(\{S_1, S_2, \dots, S_\ell\}) = \text{ebwt}(\{s_{1,1}, \dots, s_{1,m_1}, \dots, s_{\ell,1}, \dots, s_{\ell,m_\ell}\})$ ,  $\text{lcp}(\mathcal{S}) = \text{lcp}(\{S_1, S_2, \dots, S_\ell\}) = \text{lcp}(\{s_{1,1}, \dots, s_{\ell,m_\ell}\})$  and  $\text{da}(\mathcal{S}) = \text{da}(\{S_1, S_2, \dots, S_\ell\}) = \text{da}(\{s_{1,1}, \dots, s_{\ell,m_\ell}\})$ . These data structures are well known in string processing, so their construction is outside the scope of this paper (see Section 4 for a collection of methods and tools to construct them).



We extend the above, commonly used, notion of DA to *Color Document Array* (CDA), where  $\text{cda}(\mathcal{S})[i] = r$  if  $\text{da}(\mathcal{S})[i] = u$  and  $s_u$  belongs to the set  $S_r$ . In other words, we assign a same color to the strings belonging to the same set  $S_r$ , so we have a distinct color  $r$  for each set  $S_r \in \mathcal{S}$ .

► **Example 5** (running example). In Figure 1,  $\text{cda}$  coincides with  $\text{da}$  assuming that each organism is a single string.

Let  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , where each  $Q_i$  represents a node of the phylogeny tree, and thus, it identifies a non-empty set of organisms. We define a map  $\chi_{\mathcal{Q}}$  that associates any organism to the element of  $\mathcal{Q}$  to which it belongs (if there exists).

► **Definition 6.** Given  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , we define  $\chi_{\mathcal{Q}}$  from  $\{1, \dots, \ell\}$  to  $\{Q_1, \dots, Q_q\} \cup \{\emptyset\}$ , such that

$$\chi_{\mathcal{Q}}(r) = \begin{cases} Q_s & \text{if there exists } Q_s \in \mathcal{Q} \text{ s.t. } r \text{ belongs to } Q_s \\ \emptyset & \text{otherwise.} \end{cases}$$

Recall that we denote by  $\text{eBWTCLUST}[i, j]$  the concatenation of the symbols in the eBWT associated with the range  $[i, j]$  (i.e.  $\text{ebwt}(\mathcal{S})[i, j]$ ), where  $[i, j]$  is a positional cluster, unless otherwise specified.

Then, for each  $\text{eBWTCLUST}[i, j]$ , the corresponding interval in the CDA,  $\text{cda}(\mathcal{S})[i, j]$ , determines the organisms (or colors) to which the symbols in  $\text{eBWTCLUST}[i, j]$  belong.

► **Definition 7.** An  $\text{eBWTCLUST}[i, j]$  is  $\gamma_{\mathcal{Q}}$ -colored if  $\gamma_{\mathcal{Q}}$  is the set of elements of  $\mathcal{Q}$  appearing in  $\text{cda}(\mathcal{S})[i, j]$ , i.e.  $\gamma_{\mathcal{Q}} = \{\chi_{\mathcal{Q}}(r) : r \in \text{cda}(\mathcal{S})[i, j]\}$ .

Note that if  $\text{eBWTCLUST}$  and CDA are restricted to the strings in  $\mathcal{Q}$  (see Remark 1), then  $\gamma_{\mathcal{Q}}$  contains only non-empty sets.

► **Example 8.** Let  $\mathcal{Q} = \{\{S_1, S_3, S_4\}, \{S_2\}, \{S_5\}\}$  and  $\text{eBWTCLUST}[i, j] = \text{ACAAGT}$  with  $\text{cda}[i, j] = [1\ 2\ 1\ 1\ 3\ 4]$ . Then,  $\text{eBWTCLUST}[i, j]$  is  $\gamma_{\mathcal{Q}}$ -colored and  $\gamma_{\mathcal{Q}} = \{\{S_1, S_3, S_4\}, \{S_2\}\}$ .

The main idea is to detect and analyze only eBWT positional clusters associated with left-maximal contexts shared by some  $Q_i$ :

► **Definition 9.** A  $\gamma_{\mathcal{Q}}$ -colored  $\text{eBWTCLUST}[i, j]$  is *relevant*, if  $\text{ebwt}[i, j]$  is not a concatenation of a same symbol (i.e. it is not a run) and  $1 < \text{card}(\gamma_{\mathcal{Q}}) < q$  holds.

► **Example 10** (running example). We highlight in bold, in Figure 1, the relevant  $\text{eBWTCLUST}$ , that are  $\text{eBWTCLUST}[11, 14]$  and  $\text{eBWTCLUST}[22, 23]$ . Every other  $\text{eBWTCLUST}$  is either a run of a same symbol or the associated  $\text{cda}$  contains only one color or all of them.

Now, we use the notion of relevant  $\text{eBWTCLUST}$  to obtain a partition  $\mathcal{P} = \{P_1, \dots, P_p\}$  of disjoint subsets from  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ , with  $q \geq p > 1$ . Recall that  $p > 1$ , because the trivial partition  $\mathcal{P} = \{\mathcal{Q}\}$  provides no significant information. The partitioning strategy is summarized in the following three phases:

1. we scan our data structures computed on  $\mathcal{S}$ , and we detect the relevant  $\text{eBWTCLUST}$  of  $\text{eBWT}(\mathcal{S})|_{\mathcal{Q}}$  (denoted by  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$ , for some  $i < j$ );
2. we analyze each  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$  in order to build a list  $\mathcal{L}(\mathcal{Q})$  of subsets of  $\mathcal{Q}$ , called *candidate parts*, and we incrementally assign a score to each candidate part;
3. we build  $\mathcal{P}$  from the list  $\mathcal{L}(\mathcal{Q})$  by selecting *compatible* candidates with the highest score.

Note that, by using Remark 1, it is easy to verify that the relevant  $\text{eBWTCLUST}_{\mathcal{Q}}$  of the first step can be obtained by a linear scan of the input data structures.

At step 2), while analyzing any relevant  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$ , we require that any element  $Q_s$  in  $\gamma_{\mathcal{Q}}$  is a representative, *i.e.* the number of colors of  $Q_s$  appearing in  $[i, j]$  is sufficiently large. We denote by  $\tau$  ( $0 < \tau \leq 1$ ) such support threshold that determines the minimum required portion for each  $Q_s \in \gamma_{\mathcal{Q}}$ . Intuitively, the support threshold guarantees that all the elements of  $\mathcal{Q}$  appearing in the  $\text{eBWTCLUST}$  are sufficiently represented. In fact, when the support threshold approaches 1, all the elements of the subset  $Q_s$  considered are required to be in the cluster. In other words, we aim at measuring how similar the shared history of the phylogeny is in terms of shared substrings. On the other hand, when the support threshold approaches 0, at least one of the elements of the subset  $Q_s$  is required to be in the cluster considered. Thus, we are observing how similar all the evolution events are, providing two different viewpoints of their phylogeny. More formally,

► **Definition 11.** Let  $\mathcal{Q} = \{Q_1, \dots, Q_q\}$  and  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$  be relevant. Given a support threshold value  $\tau$  in  $(0, 1]$ , we define the *candidate part* of  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$ , and we denote it by  $L_{[i, j]}$ , the set  $\gamma_{\mathcal{Q}}$  only if it holds  $\text{card}(\text{cda}[i, j] \cap Q_s) \geq \tau \cdot \text{card}(Q_s)$ , for all  $Q_s \in \gamma_{\mathcal{Q}}$ .

In general, any relevant  $\gamma_{\mathcal{Q}}$ -colored  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$  may not have an associated candidate part  $L_{[i, j]}$ . That is the case in which some  $Q_s \in \gamma_{\mathcal{Q}}$  are not sufficiently represented in the interval  $[i, j]$ .

The list  $\mathfrak{L}(\mathcal{Q})$  of all candidate parts  $L_{[i, j]} \subset \mathcal{Q}$  is built up by analyzing all  $\text{eBWTCLUST}_{\mathcal{Q}}$ . Any  $\text{eBWTCLUST}_{\mathcal{Q}}[i, j]$  contributes to the score of its candidate part  $L_{[i, j]} \subset \mathcal{Q}$  by the minimum value in  $\text{lcp}[i + 1, j]$ . So, the elements of the list  $\mathfrak{L}(\mathcal{Q})$  appear as pairs  $(L, y)$ , where  $L$  is a proper subset of  $\mathcal{Q}$  and  $y$  its associated score incrementally obtained.

Intuitively, we use the score to determine the order in which the candidate parts must be taken into account to build  $\mathcal{P}$ . Since  $\mathcal{P}$  is a partition of  $\mathcal{Q}$ , we cannot take all the candidate parts with a high score, but we must select, step by step, only those that are somehow compatible with each other. In fact, by partition definition, if  $X$  is a part of  $\mathcal{P}$  and  $Y$  has non-empty intersection with  $X$ , then  $Y$  cannot be a part of  $\mathcal{P}$ .

► **Example 12 (running example).** Let us consider the first call of PARTITION for the toy example in Figure 1, where  $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$  and  $Q_i = \{S_i\}$ . Let  $\tau$  be any value in  $(0, 1]$ ,  $\text{eBWTCLUST}[11, 14]$  contributes to increase the score of its candidate part  $\{Q_1, Q_3\}$  by 1 (being the minimum lcp-value in  $[12, 14]$ ), while  $\text{eBWTCLUST}[22, 23]$  contributes to increase the score of its candidate part  $\{Q_1, Q_2\}$  by 6. So, at the end of the cluster analysis, we have that  $\mathfrak{L}(\mathcal{Q})$  contains  $(\{Q_1, Q_2\}, 6)$ ,  $(\{Q_1, Q_3\}, 1)$ . The output partition  $\mathcal{P}$  is  $\{\{Q_1, Q_2\}, \{Q_3\}\}$ , as  $\{Q_1, Q_2\}$  has a higher score than  $\{Q_1, Q_3\}$ .

We use a greedy algorithm to select a list (denoted by  $L_C$ ) of compatible candidate parts that will constitute the parts of the output partition  $\mathcal{P}$ .

In our selecting procedure, we choose to consider only elements of  $\mathfrak{L}(\mathcal{Q})$  having high scores. In particular, we denote by  $\bar{t}$  the number of elements with the highest scores such that the difference with the previous higher score does not form a local minimum and at least  $t$  highest scores are considered.

In addition to the list  $L_C$  of compatible subsets, during the scan of such  $\bar{t}$  subsets with the highest scores, we build a second list (denoted by  $L_E$ ) of *compatible extensions* of  $L_C$ .

Both lists  $L_C$  and  $L_E$  are initially empty. A subset  $V$  is compatible, and we add it to  $L_C$ , if it is disjoint from all the elements already in  $L_C$ , and moreover, if either it has empty intersection with all the elements in the list  $L_E$ , or it is strictly contained in the first element with non-empty intersection. We define a set  $V$  compatible extension, and we add it to  $L_E$ , if it is a superset of all the elements already in  $L_C$ .

Intuitively, the list  $L_E$  contains those subsets that warn us from selecting next compatible subsets that separate its elements. In fact, the order in which subsets are processed is given by their score, so when checking if  $V$  is compatible, all the elements already in  $L_C$  and in  $L_E$  show a score higher than  $V$ .

The output partition  $\mathcal{P}$  contains all the compatible subsets in  $L_C$  as parts, and any other element of  $\mathcal{Q}$  not appearing in any subset in  $L_C$ , as a singleton part.

### 3.3 Complexity

**Partitioning.** We observe here how the partitioning procedure described in Subsection 3.2 can be computed in  $O(N)$  time and space, where  $N$  corresponds to the sum of lengths of all strings within the organisms in  $\mathcal{Q}$ . Indeed, the eBWT can be computed in linear time in its length, which is  $N$ , including the identification of all positional clusters [28]. Given an element of a eBWTCLUST $[i, j]$  and  $\tau$ , we can determine in  $O(1)$  time its color using the CDA; as we can pre-compute the size of all  $Q_i$ , this lets us easily determine the  $\gamma_{\mathcal{Q}}$ -coloring of the cluster and the associated  $L_{[i,j]}$  (Definitions 7 and 11) in time proportional to the cluster's length, for a total cost of  $O(N)$  over the whole BWT to obtain  $\mathcal{L}(\mathcal{Q})$ .

While potentially there could be up to  $2^g \leq 2^\ell$  candidate parts  $L_{[i,j]}$ , observe that each positional cluster can in fact define at most one candidate, of size not greater than the length of the cluster; it follows that  $\mathcal{L}(\mathcal{Q})$  has  $< N$  elements, and the sum of sizes of all  $L_{[i,j]}$  is too at most  $N$ .

Next, the algorithm sorts  $\mathcal{L}(\mathcal{Q})$  by score, which using a bucket sort takes  $O(N)$  time. Finally, we need to scan  $\mathcal{L}(\mathcal{Q})$  to obtain  $L_C$ : using bit-vectors (or other standard data structures) we can keep track in constant time of which  $Q_i$  have been added to  $L_C$  or to  $L_E$ ; thus we can check if an  $L_{[i,j]}$  is compatible (or if it generates a compatible extension) in time proportional to its size, meaning the total cost of this step is once more  $O(N)$ , and so are the running time and space requirements of the partitioning procedure.

**Tree reconstruction.** While we omit a more detailed analysis, it is relatively straightforward to see that each  $\mathcal{Q}$  placed in the queue corresponds to one node of the tree (see Line 10 in Algorithm 1).

Since PARTITION always splits the input in at least two parts, the number of internal nodes are at most the number of leaves ( $\ell$ ); as such, the complexity is bounded by  $\ell$  times the cost of PARTITION. As described above, the latter is bound by the sum of lengths of the strings in the  $\mathcal{Q}$  PARTITION is called upon, which is at most the size of the input  $O(N)$  (although this is a worst case). As for the space requirement, it is that of PARTITION plus the maximum size of the queue: the queue holds up to  $O(\ell)$  elements (one for each node of the tree), and each has size at most  $\ell$ . The following holds:

► **Lemma 13.** *Given a set  $\mathcal{S}$  of  $\ell$  organisms, whose total length is  $N$ , phyBWT reconstructs a phylogenetic tree for  $\mathcal{S}$  in  $O(N\ell)$  time and  $O(N + \ell^2)$  space.*

We observe that  $N$  is the dominant factor in this complexity, as the length of the strings representing a taxon is -in known applications- many orders of magnitude greater than the number  $\ell$  of taxa. Moreover, letting  $n = N/\ell$  be the average length of a taxon, the time cost  $O(N\ell)$  can be equivalently seen as  $O(n\ell^2)$ , so quadratic in the number of taxa.

## 4 Preliminary experiments

In this section we assess our partitioning-based method, phyBWT, for reconstructing phylogenetic trees from short-reads and *de novo* assembled sequences. Indeed, phyBWT is not limited to a particular type of input, and it is able to manage both types of data. However, the diversity in the type of input data needs a tuning of the parameters described in the method section.

In the literature, features similar to phyBWT are shared by the tool SANS [38, 31] which, as our tool, is an alignment- and reference-free approach that is whole-genome based, and in addition, it does not produce a pairwise comparison of the sequences or their characteristics. Differently from phyBWT, SANS is based on the computation of all  $k$ -mers, which are either directly extracted or read in a colored de Bruijn graph, and then used to build a list of splits. The first implementation of SANS [38] post-processed the list of splits according to two filtering strategies that are described in the software tool SplitsTree [20]: (i) a greedy weakly approach that allows to display the output as a network, and (ii) a greedy tree approach that displays the output as a tree. In fact, according to the phylogenetic splits model [3], the reconstructed phylogenies are mesh-like graph and they are not restricted to trees.

The latest version<sup>3</sup> of the tool SANS [31] is a stand-alone re-implementation that introduces some new features and improves the runtime and the memory usage. It has mainly three filtering options that allow to limit the output splits in order to reduce the complexity of the network or calculate a subset of the splits representing a tree.

We show experiments carried out by such new version of the tool SANS by applying the filtering approach for drawing trees.

**Drawing phylogeny tree.** Our tool reconstructs a tree by means of partitions and outputs can be visualized by using well-known existing tools. In this paper, all the trees are drawn by using the PHYLIP package by Joe Felsenstein<sup>4</sup> and manually annotated.

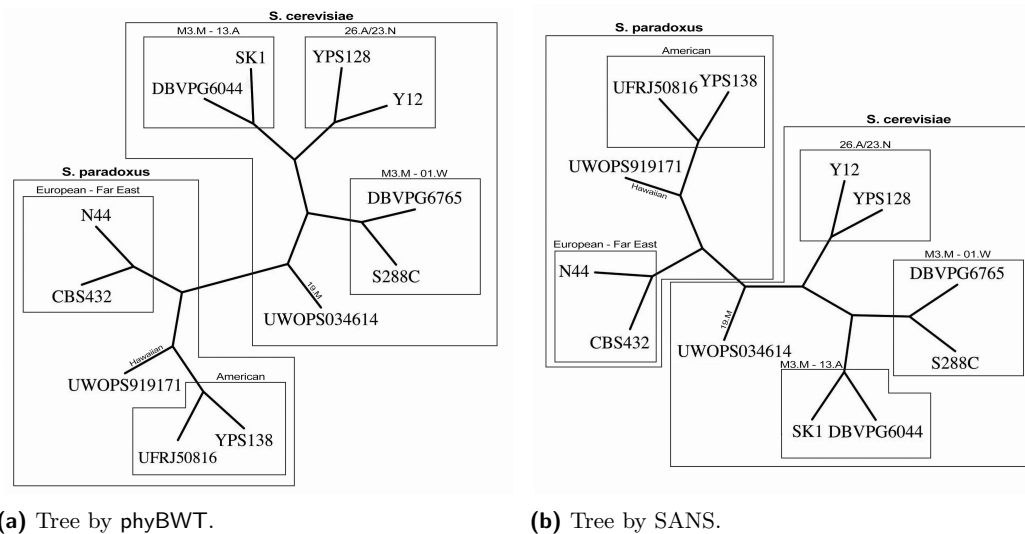
**Data Structure building.** We observe that we take in input the following data structures eBWT, LCP and DA that can be built independently from our tool, *e.g.* [5, 10, 13, 7, 23, 30, 8]. One also could build them for each  $S_i$ , then, one could merge them to get the data structures of the union of some  $S_i$ .

**Datasets.** To show the effectiveness of our method, we have chosen three datasets with a similar number of organisms but with a diverse composition and different length of the strings. More in details, we used three different types of datasets: i) Illumina sequencing data (short reads) for seven *S. cerevisiae* and five *S. paradoxus* strains from the study in [41]; ii) assemblies from 12 species of the genus *Drosophila* from the FlyBase database (largely accepted phylogeny shown in [11]); iii) viral complete genomes from the *Prasinovirus* genus (benchmark phylogeny trees reported in [15]). The datasets ii) and iii) are also analyzed in [38].

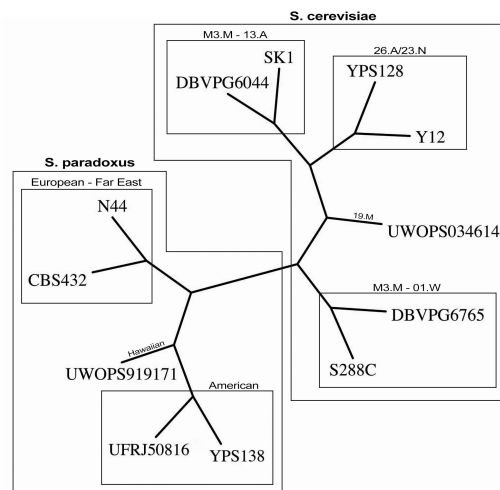
**Running time.** While our implementation of phyBWT is not yet optimized, we observe that the running time of our prototype is dominated by the cost of computing eBWT, LCP and DA. When the latter cost is stripped down, the running time of phyBWT is lower compared

<sup>3</sup> <https://gitlab.ub.uni-bielefeld.de/gi/sans>

<sup>4</sup> <https://evolution.genetics.washington.edu/phylip/>, version 3.698 for 64-bit Windows systems.



■ **Figure 3** Yeasts phylogeny by phyBWT (a) and by SANS (b).

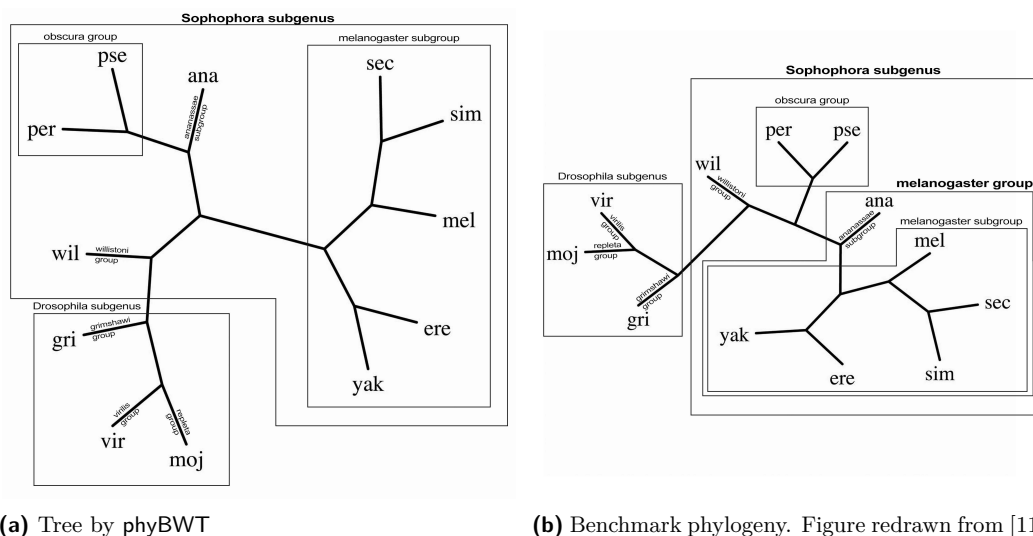


■ **Figure 4** Benchmark phylogeny for the yeasts dataset. Figure redrawn from [41].

to that of SANS. However, as it is often the case, eBWT, LCP and DA (or a subset of them) might be already available from some other applications. For example, for the medium-size *Drosophila* dataset, phyBWT infers the phylogenetic tree in 2 minutes by using 25 GB of internal memory, and SANS runs in 26 minutes using 28 GB of internal memory.

#### 4.1 Yeasts dataset

This dataset comprises 12 Illumina sequencing experiments obtained from the study in [41], and deposited in the public repository SRA (Short Reads Archive) under accession code PRJNA340312. The 12 datasets from [41] include seven sequencing experiments for the *S. cerevisiae* strains and five for the *S. paradoxus* strains. According to [41], for each sequencing experiment, comprising 151-bp paired-end reads, we performed adaptor-removing and quality-based trimming using trimmomatic [6]. For each sample, we extracted 5 million of 151-bp paired-end reads to form the yeasts dataset with a total size of 26 GB.



■ **Figure 5** Drosophila phylogeny: (a) by our method; (b) benchmark redrawn from [11].

We ran both phyBWT and SANS on such short reads dataset. The tree depicted in Figure 3a is produced by phyBWT for any  $k_m > 13$ ,  $\tau = 0.6$  and  $t = 12$  (as the number of taxa). For SANS, we used default parameters that corresponds to a  $k$ -mer length of 31, and we set the parameter `-f strict` to output a tree in the Newick format (see Figure 3b).

The benchmark tree reported in Figure 4 is the one obtained in the original study [41]. Remarkably, the benchmark was built using nuclear one-to-one orthologs, i.e. blocks of nuclear genes which are shared among (1) the seven *S. cerevisiae*, (3) the five *S. paradoxus* strains sequenced in the study, and (3) six outgroups from the *Saccharomyces* genus.

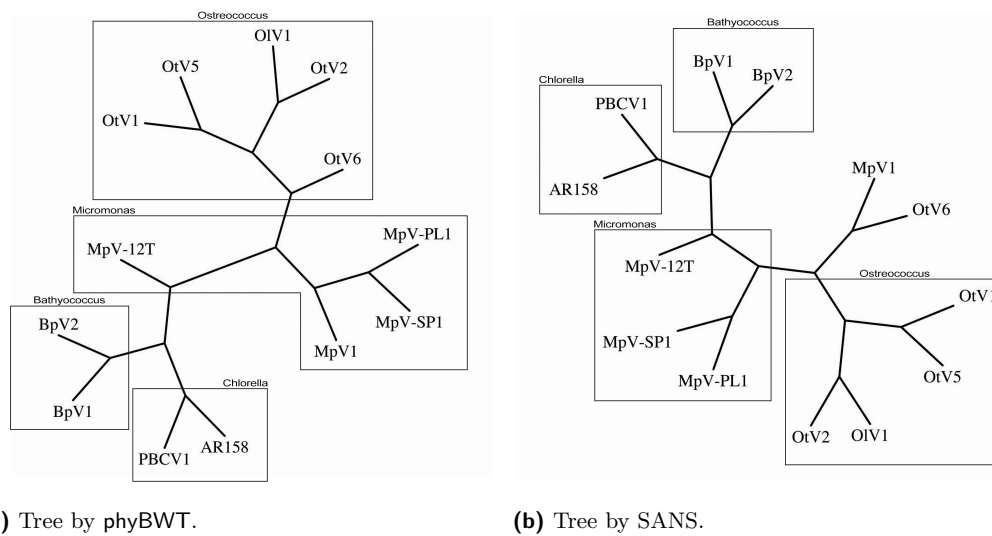
Both phyBWT and SANS correctly group the *S. cerevisiae* and the *S. paradoxus* strains which show an average whole-genome sequence divergence of  $\sim 10\%$ . As expected by taking into account the relatively high divergence among *S. paradoxus* strains (0.5% - 4.5%), also the same *S. paradoxus* partition is obtained. On the other hand, a few differences are shown in the *S. cerevisiae* partition which groups strains with a sequence divergence  $\sim 0.5\%$ . Compared to SANS, phyBWT produces a tree which is closer to the benchmark although the differences with the benchmark shown by both SANS and our method can be explained considering the relatively low divergence among *S. cerevisiae* strains as well as the partially admixed genomes of some of the trains (e.g. S288C and DBVPG6044) [27].

## 4.2 Drosophila dataset

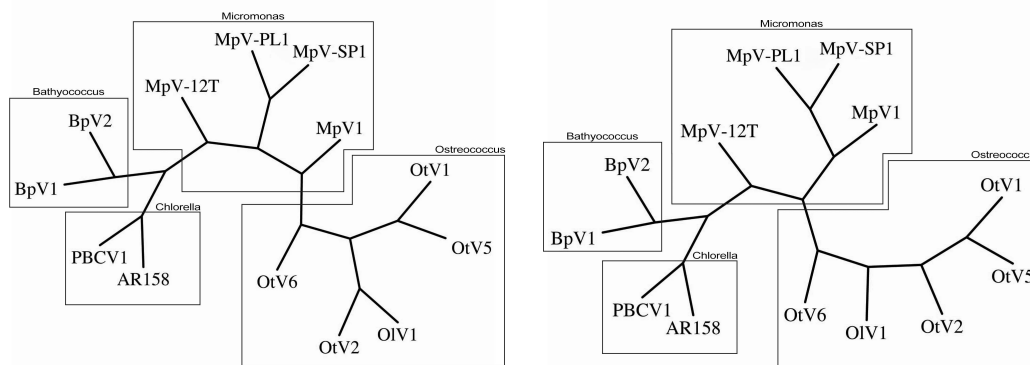
Drosophila data are downloaded from the FlyBase database (<http://flybase.org/>). This dataset includes assemblies from 12 species of the genus *Drosophila*: *D. melanogaster* (mel), *D. ananassae* (ana), *D. erecta* (ere), *D. grimshawi* (gri), *D. mojavensis* (moj), *D. persimilis* (per), *D. pseudoobscura* (pse), *D. sechellia* (sec), *D. simulans* (sim), *D. virilis* (vir), *D. willistoni* (wil), and *D. yakuba* (yak). Nine of these species fall within the Sophophora subgenus, which includes members of the melanogaster, obscura and willistoni groups.

The number of strings for each species varies: it ranges from 1,870 for *D. melanogaster* to 17,440 for *D. grimshawi*. The obtained dataset is a medium-sized input with a total number of symbols of more than 2,161 Mbp.





■ **Figure 6** *Prasinovirus* by phyBWT and by SANS.



■ **Figure 7** Benchmark phylogeny for *Prasinovirus* dataset. Figures redrawn from [15].

phyBWT produces the tree depicted in Figure 5a for any  $k_m$  in  $[23, 72]$ ,  $\tau = 0.5$  and  $t = 12$  (as the number of taxa). The *Sophophora* subgenus as well as the *Drosophila* subgenus are correctly detected, and inside the *Sophophora* subgenus, the *melanogaster* subgroup is correctly isolated. The only difference with respect to the benchmark tree by [11] is the organism *D. ananassae* that represents the *ananassae* subgroup. Such subgroup is part of the *melanogaster* group together with *D. melanogaster*, *D. sechellia*, *D. simulans*, *D. erecta* and *D. yakuba*. However, our method places *D. ananassae* closer to the *obscura* group rather than the *melanogaster* subgroup. SANS was run with default values as described in [38]. The output tree obtained by setting `-f strict` is topological equivalent to the benchmark reference tree in [11], which has reported in Figure 5b for completeness.

### 4.3 *Prasinovirus* dataset

This dataset comprises 13 genomes from the viral genus *Prasinovirus* studied in [15] and infecting the genera *Ostreococcus* (OtV1, OtV2, OtV5, OtV6, OIV1), *Bathycoccus* (BpV1, BpV2), *Micromonas* (MpV1, MpV-12T, MpV-PL1, MpV-SP1), and *Chlorella* (PBCV1, AR158).

Phylogenetic reconstruction of viral genomes is challenging due to their small genome size and the high variability of their genome content.

In [15], the authors reported two different phylogenetic trees: one is based on the presence/absence of shared putative genes [15, Figure 3] and the other is a maximum likelihood estimation based on a marker gene (DNA polymerase B) [15, Figure 4]. The two benchmark trees are depicted in Figure 7.

The complete prasinovirus genomes used in this dataset are 213 Kbp on average: in particular, the genome sizes range from 173,350 bp for MpV-SP1 to 205,622 bp for MpV-12T.

phyBWT produces the tree depicted in Figure 6a for  $k_m = 13$ ,  $\tau = 0.5$  and  $t = 13$  (as the number of the taxa), while Figure 6b depicts the output tree by SANS generated by using parameters `-k 11 -t 130 -f strict` (recommended parameter for such dataset, see <https://gitlab.uni-bielefeld.de/gi/sans>).

The only main difference of the tree produced by SANS with respect to the benchmark trees is that the former shows a subtree with MpV1 and OtV6 as leaves, although MpV1 belongs to *Micromonas* group and OtV6 belongs to *Ostreococcus* group.

## 5 Conclusions and discussion

In this paper, we proposed a new alignment-, assembly- and reference-free partition-based method to build the phylogeny inference of a set of organisms.

To the best of our knowledge, phyBWT is the first method that applies the properties of the Extended Burrows-Wheeler Transform (eBWT) to the idea of decomposition for phylogenetic inference. Our approach is based on the eBWT positional cluster framework introduced in [29], which allowed us to consider longest shared substrings of varying length, unlike  $k$ -mer-based approaches such as SANS. We introduce the inner partitioning algorithm based on the eBWT positional cluster, and employ it as a black-box in our novel tree reconstruction algorithm. Specifically, phyBWT does not start from the leaves to group them together in a bottom-up fashion; it does not start from the root and performs a top-down partition; instead, it proceeds in both directions (bottom-up and top-down), according to what is returned by the inner partitioning algorithm.

We tested our method on three sequencing datasets, with short reads and *de novo* assembled sequences. The experimental results show that our algorithm produces trees comparable to the benchmark phylogeny and to the newly introduced tool SANS. We plan to perform algorithm engineering of phyBWT that will better exploit the bounded length of the reads to overcome the computational bottleneck of computing the eBWT.

While the worst-case complexity of the method is competitive with existing methods, there are interesting directions for further optimization, such as using Colored Range Queries [16] to speed up identification of colors in the various clusters, or exploiting the natural predisposition of the method for parallel computation. Our current prototype requires a preprocessing in order to compute some data structures that are at the heart of several text and string algorithms. More efficient tools for computing them can appear in the literature, and we plan to investigate whether the required information can be reduced for instances of our problem.

## References

- 1 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.
- 2 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.
- 3 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.
- 4 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/>.
- 5 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.
- 6 A. M Bolger, M. Lohse, and B. Usadel. Trimmomatic: a flexible trimmer for illumina sequence data. *Bioinformatics*, 30(15):2114–20, 2014. doi:10.1093/bioinformatics/btu170.
- 7 P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, and R. Rizzi. Multithread Multistring Burrows-Wheeler Transform and Longest Common Prefix Array. *Journal of computational biology*, 26(9):948–961, 2019. doi:10.1089/cmb.2018.0230.
- 8 C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, and M. Sciortino. Computing the original ebwt faster, simpler, and with less memory. In *SPIRE*, pages 129–142. Springer International Publishing, 2021.
- 9 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.
- 10 A.J. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37:17–33, 2016. doi:10.1016/j.jda.2016.03.003.
- 11 M. A. Crosby, J. L. Goodman, V. B. Strelets, P. Zhang, W. M. Gelbart, and The FlyBase Consortium. FlyBase: genomes by the dozen. *Nucleic Acids Research*, 35(suppl.1):D486–D491, November 2006.
- 12 M. D’Angiolo, M. De Chiara, J.-X. Yue, A. Irizar, S. Stenberg, K. Persson, A. Llored, B. Barré, J. Schacherer, R. Marangoni, E. Gilson, J. Warringer, and G. Liti. A yeast living ancestor reveals the origin of genomic introgressions. *Nature*, 587(7834):420–425, November 2020.
- 13 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.
- 14 H. Fan, A. R Ives, Y. Surget-Groba, and C. H Cannon. An assembly and alignment-free method of phylogeny reconstruction from next-generation sequencing data. *BMC genomics*, 16(1):1–18, 2015.
- 15 J. F. Finke, D. M. Winget, A. M. Chan, and C. Suttle. Variation in the genetic repertoire of viruses infecting micromonas pusilla reflects horizontal gene transfer and links to their environmental distribution. *Viruses*, 9, 2017.
- 16 Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013. doi:10.1016/j.tcs.2012.08.004.
- 17 B. Gallone, J. Steensels, S. Mertens, M. C. Dzialo, J. L. Gordon, R. Wauters, F. A. Theßeling, F. Bellinazzo, V. Saels, B. Herrera-Malaver, T. Prahl, C. White, M. Hutzler, F. Meußdoerffer, P. Malcorps, B. Souffriau, L. Daenen, G. Baele, S. Maere, and K. J. Verstrepen. Interspecific hybridization facilitates niche adaptation in beer yeast. *Nat Ecol Evol*, 3(11):1562–1575, November 2019.

- 18 V. Guerrini, F. Louza., and G. Rosone. Lossy Compressor Preserving Variant Calling through Extended BWT. In *BIOSTEC/BIOINFORMATICS*, pages 38–48. INSTICC, SciTePress, 2022. doi:10.5220/0010834100003123.
- 19 V. Guerrini, F.A. Louza, and G. Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinformatics*, 21, 2020. doi:10.1186/s12859-020-03628-w.
- 20 D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, October 2005.
- 21 J. Jansson and W.-K. Sung. *Algorithms for Combining Rooted Triplets into a Galled Phylogenetic Network*, pages 48–52. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4\_92.
- 22 J. Jansson and W.-K. Sung. *Maximum Agreement Supertree*, pages 1224–1227. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4\_222.
- 23 F. A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and G. Rosone. gsufsort: constructing suffix arrays, lcp arrays and bwts for string collections. *Algorithms for Molecular Biology*, 15, 2020.
- 24 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *ACM-SIAM SODA*, pages 319–327, 1990.
- 25 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theoret. Comput. Sci.*, 387(3):298–312, 2007.
- 26 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. A new combinatorial approach to sequence comparison. *Theory Comput. Syst.*, 42(3):411–429, 2008. doi:10.1007/s00224-007-9078-6.
- 27 J. Peter, M. De Chiara, A. Friedrich, J.-X. Yue, D. Pflieger, A. Bergström, A. Sigwalt, B. Barre, K. Freel, A. Llored, C. Cruaud, K. Labadie, J.-M. Aury, B. Istace, K. Lebrigand, P. Barbry, S. Engelen, A. Lemainque, P. Wincker, and J. Schacherer. Genome evolution across 1,011 *saccharomyces cerevisiae* isolates. *Nature*, 556, April 2018. doi:10.1038/s41586-018-0030-5.
- 28 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms for Molecular Biology*, 14(1):3, 2019. doi:10.1186/s13015-019-0137-8.
- 29 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.
- 30 N. Prezza and G. Rosone. Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138–156, 2021. doi:10.1016/j.tcs.2020.11.024.
- 31 A. Rempel and R. Wittler. SANS serif: alignment-free, whole-genome-based phylogenetic reconstruction. *Bioinformatics*, 37(24):4868–4870, 2021. doi:10.1093/bioinformatics/btab444.
- 32 N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425, 1987.
- 33 S. M Soucy, J. Huang, and J. P. Gogarten. Horizontal gene transfer: building the web of life. *Nat Rev Genet*, 16(8):472–82, August 2015.
- 34 L. Tattini, N. Tellini, S. Mozzachiodi, M. D’Angiolo, S. Loeillet, A. Nicolas, and G. Liti. Accurate tracking of the mutational landscape of diploid hybrid genomes. *Mol Biol Evol*, August 2019.
- 35 S. Vinga. Alignment-free methods in computational biology, 2014.
- 36 S. Vinga and J. Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 2003. doi:10.1093/bioinformatics/btg005.
- 37 T. Warnow. *Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation*. Cambridge University Press, 2017.
- 38 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 fuer Informatik. doi:10.4230/LIPIcs.WABI.2019.2.

- 39 L. Yang, X. Zhang, and T. Wang. The Burrows-Wheeler similarity distribution between biological sequences based on Burrows-Wheeler transform. *Journal of Theoretical Biology*, 262(4):742–749, 2010. doi:10.1016/j.jtbi.2009.10.033.
- 40 Z. Yang and B. Rannala. Molecular phylogenetics: Principles and practice. *Nature reviews. Genetics*, 13:303–14, March 2012. doi:10.1038/nrg3186.
- 41 J.-X. Yue, J. Li, L. Aigrain, J. Hallin, K. Persson, K. Oliver, A. Bergström, P. Coupland, J. Warringer, M. C. Lagomarsino, G. Fischer, R. Durbin, and G. Liti. Contrasting evolutionary genome dynamics between domesticated and wild yeasts. *Nature Genetics*, 49(6):913–924, 2017. doi:10.1038/ng.3847.
- 42 A. Zielezinski, S. Vinga, J. Almeida, and W. Karlowski. Alignment-free sequence comparison: Benefits, applications, and tools. *Genome Biology*, 18:186, October 2017. doi:10.1186/s13059-017-1319-7.