The Author(s) *BMC Bioinformatics* 2017, **18**(Suppl 3):59
DOI 10.1186/s12859-017-1466-6

BMC Bioinformatics

RESEARCH Open Access

A framework for space-efficient read clustering in metagenomic samples

Jarno Alanko[1*], Fabio Cunial[2], Djamal Belazzougui[3] and Veli Mäkinen[1]

From The Fifteenth Asia Pacific Bioinformatics Conference
Shenzhen, China. 16-18 January 2017

Abstract

Background: A metagenomic sample is a set of DNA fragments, randomly extracted from multiple cells in an environment, belonging to distinct, often unknown species. Unsupervised metagenomic clustering aims at partitioning a metagenomic sample into sets that approximate taxonomic units, without using reference genomes. Since samples are large and steadily growing, space-efficient clustering algorithms are strongly needed.

Results: We design and implement a space-efficient algorithmic framework that solves a number of core primitives in unsupervised metagenomic clustering using just the bidirectional Burrows-Wheeler index and a union-find data structure on the set of reads. When run on a sample of total length n, with m reads of maximum length ℓ each, on an alphabet of total size σ, our algorithms take $O(n(t + \log \sigma))$ time and just $2n + o(n) + O(\max\{\ell\sigma \log n, K \log m\})$ bits of space in addition to the index and to the union-find data structure, where K is a measure of the redundancy of the sample and t is the query time of the union-find data structure.

Conclusions: Our experimental results show that our algorithms are practical, they can exploit multiple cores by a parallel traversal of the suffix-link tree, and they are competitive both in space and in time with the state of the art.

Keywords: Metagenomics, Read clustering, Text indexing, Burrows-Wheeler transform, Suffix-link tree, Right-maximal *k*-mer, Submaximal repeat

Background

High-throughput sequencing has made it fast and cost-effective to sequence DNA from entire environments at once. The collection of all genomes in an environment is called the *metagenome* of the environment. A fundamental problem in metagenomics is to cluster the reads produced by a high-throughput experiment, according to which species (or, more generally, taxonomic unit) they originate from. This can be done in a supervised manner, by mapping the reads to a database of known genomes, or in an unsupervised way, by performing extensive comparisons of all reads against each other without relying on any reference database. Unsupervised methods are attractive, and in most practical cases the only option available, since the genome of most organisms (e.g. prokaryotes) that inhabit complex environments is unknown.

Having accurate clusters for reads that come from unknown taxonomic units allows one to estimate key measures of environmental biodiversity, and to assemble the corresponding genomes more accurately and using less memory [1–3]. Clusters have also natural applications to comparative genomics, as well as to the emerging field of *comparative metagenomics* that is becoming increasingly crucial for managing and understanding collections of hundreds of thousands of samples, like those already available in [4, 5]. For example, a cluster corresponding to an unknown taxonomic unit could be positioned inside a taxonomy of known genomes by comparing their substring composition, and two metagenomic samples with annotated clusters could be compared in time proportional to the number of clusters, for example using the measures described in [6], rather than in time proportional to the number of distinct substrings of a specific

*Correspondence: jarno.alanko@cs.helsinki.fi
[1]Department of Computer Science, University of Helsinki, Gustaf Hällströmin katu 2b, 00560 Helsinki, Finland
Full list of author information is available at the end of the article

length, as done e.g. in [7, 8], or to the number of reads, as done in [9].

The problem of building a scalable, accurate and unsupervised metagenomic clustering pipeline is fairly recent and still open. Most existing approaches exploit the same signal as alignment-free genome comparison tools, namely taxon-specific biases in the frequency of substrings of a specific length, and they use such signal in hidden Markov models [10, 11], maximum likelihood Monte Carlo Markov chains [12], and expectation maximization algorithms [11, 13]. Other pipelines blend statistics with combinatorial criteria, merging e.g. reads that share long substrings first, and then further merging such groups by similarity of their k-mer composition vectors (see e.g. [14–16] and references therein). A single metagenomic sample contains tens of gigabases, and to improve accuracy it is becoming more common to cluster *the union of multiple samples* that are believed to contain shared taxonomic units, as described e.g. in [17, 18]. Notwithstanding such issues of scale, no existing clustering pipeline is designed to be space-efficient, and can thus handle with commodity hardware the largest datasets available.

Read clustering

Read clustering tools share a number of core combinatorial primitives, which they blend with statistical considerations and with ad hoc heuristics to achieve accuracy. A pipeline that contains most such primitives is the one described in [15], which we summarize in what follows, pointing the reader to the original paper and to its references for statistical considerations and for criteria used to set the parameters. The same primitives recur in a number of other pipelines [14, 16, 19]. As customary, we call *k-mer* any string of length k, and we call *coverage* the average number of reads that contain a position of the genome of a taxonomic unit. We also use the term *taxon* as a synonym for taxonomic unit.

The first step of the pipeline consists in detecting and filtering out reads sampled from low-frequency taxa, since in practice the presence of such reads tends to degrade the quality of the clusters of high-frequency taxonomic units. Reads from low-frequency taxa should also be clustered with dedicated settings of the parameters. Such filtering has the additional advantage of removing reads with a large number of sequencing errors, and of reducing the size of the input to the following stages. Specifically, given integers k and $\tau > 1$, a read is filtered out iff all its distinct k-mers occur (possibly reverse-complemented) less than τ times in the read set, where τ is set according to error rate, expected coverage, and read length.

Given a DNA string r, let $\widetilde{r}$ denote its reverse complement. Consider the graph where reads are vertices and two vertices are connected by an edge iff they are related in the following sense:

Definition 1 (k-RC-relation) *Two DNA strings r_1 and r_2 are related iff there is a string α of length k such that α occurs in r_1, and α or $\widetilde{\alpha}$ occurs in r_2.*

The requirement of sharing a k-mer in either the forward or the reverse-complement orientation comes from the fact that we ignore whether a read was sampled from the forward or the reverse-complement strand of its genome. The second step of the pipeline consists in computing the connected components of the read graph defined by the k-RC-relation: we call such components the *k-RC connected components*. We omit k whenever its value is clear from the context. The value of k is typically set using statistics on the substrings of known genomes. The connected components of the k-RC-relation loosely correspond to unassembled contigs. Moreover, assuming that the genomes of distinct taxonomic units have approximately the same length, different taxonomic units should get approximately the same number of connected components. Connected components can be further merged if we have paired-end labels on the reads.

In this paper we will also consider the following relation, whose connected components are a refinement of those induced by the previous one:

Definition 2 (k-relation) *Two strings r_1 and r_2 are related iff there is a string α of length k such that α occurs in r_1 and α occurs in r_2.*

Note that there is a one-to-one correspondence between the connected components of the k-relation and the connected components of the de Bruijn graph of order k of the set of reads.

The third step of the pipeline consists in computing *composition vectors* of h-mers, where $h < k$, for every connected component: a composition vector is an array of 4^h elements, where each element corresponds to a distinct h-mer, and where the value of element α is the (normalized) frequency of string α in the connected component. Since reads can be sampled from both strands of a double-stranded DNA molecule, the frequency of an h-mer and of its reverse complement are summed, and a composition vector consists just of the distinct h-mers that are lexicographically smaller than their reverse complement.

Composition vectors are computed from connected components, rather than from single reads, since reads are typically too short for their h-mer composition to approximate the one of their corresponding genomes. Due to multiple occurrences of the same string inside the same connected component, the h-mer composition is not estimated directly from the reads in the connected component, but rather from distinct long substrings that repeat inside the connected component, specifically from distinct substrings of length at least $e > h$ which occur

(possibly reverse-complemented) at least τ' times in the connected component.

Composition vectors are finally clustered using e.g. the k-means algorithm, since connected components with similar h-mer composition are likely to correspond to long fragments of the same genome.

Strings and string indexes

Let $S[1,n]$ be a string with alphabet $\Sigma = [1,\sigma]$. For simplicity, we assume in what follows that $S[0]$ means $S[n]$, and that the substring $S[0,n]$ means $S[n,n]$. We denote by $\overline{S}$ the reverse of S. Given a bijective mapping $f : \Sigma \to \Sigma$ that defines a *complement* character for each character in Σ, we call *reverse complement* of S the string $\widetilde{S} = f(S[n]) \cdot f(S[n-1]) \cdot \cdots \cdot f(S[1])$. Unless otherwise noted, in this paper we assume f to be the natural complementarity of DNA bases, i.e. $f(\mathtt{A}) = \mathtt{T}, f(\mathtt{T}) = \mathtt{A}, f(\mathtt{C}) = \mathtt{G}$, $f(\mathtt{G}) = \mathtt{C}$, although our algorithms do not exploit this specific mapping.

We denote by $\mathcal{P}_S(\alpha)$ the set of all starting positions of a string $\alpha \in \Sigma^+$ in the circular version of S. We set $\Sigma_S^r(\alpha) = \{c \in \Sigma : |\mathcal{P}_S(\alpha c)| > 0\}$ and $\Sigma_S^\ell(\alpha) = \{c \in \Sigma : |\mathcal{P}_S(c\alpha)| > 0\}$. A *repeat* $\alpha \in \Sigma^+$ is a string that satisfies $|\mathcal{P}_S(\alpha)| > 1$. A repeat α is *right-maximal* (respectively, *left-maximal*) iff $|\Sigma_S^r(\alpha)| > 1$ (respectively, iff $|\Sigma_S^\ell(\alpha)| > 1$). A *maximal repeat* is a repeat that is both left- and right-maximal. It is well known that a maximal repeat corresponds to an equivalence class of the set of all right-maximal repeats. A *supermaximal repeat* is a maximal repeat that is not a substring of any other maximal repeat. We say that a left-maximal repeat α is *strongly left-maximal* iff there are at least two distinct characters a and b in Σ such that both $a\alpha$ and $b\alpha$ are right-maximal repeats of S. Clearly only right-maximal repeats of S can be strongly left-maximal, thus the set of strongly left-maximal repeats of S is a subset of the maximal repeats of S. Given a string α that occurs in S, we call $\lambda(\alpha)$ the number of (not necessarily proper) suffixes of α that are strongly left-maximal repeats of S, and we call $\lambda_S = \max\{\lambda(S[1,i]) : 1 \le i \le n)\}$. Clearly $\lambda_S \in O(n)$. Strong right-maximality is defined symmetrically. A string $\alpha \in \Sigma^+$ is a *reverse-complement right-maximal repeat* (*RC right-maximal* for short) of S if it is a right-maximal repeat of $S\$\widetilde{S}$, where $\$ \notin \Sigma$ is a separator: in other words, there are two distinct characters $c, d \in \Sigma$ such that αc or $\widetilde{\alpha c}$ is a substring of S, and αd or $\widetilde{\alpha d}$ is a substring of S. Reverse-complement left-maximal repeats and reverse-complement maximal repeats are defined symmetrically.

The *suffix tree* of a string $S \in [1,\sigma]^+$ is the compacted trie built on the set of all suffixes of string $S\$$, where $\$ = 0 \notin [1,\sigma]$ [20]. There is a bijection between the set of leaves of the suffix tree and the set of suffixes of $S\$$, and there is a bijection between the set of internal nodes of the suffix tree and the set of right-maximal repeats of S. We denote by $\ell(v)$ the label of a node v in the tree, i.e. the concatenation of the labels of all edges in the path from the root to v. The *locus* of a nonempty substring α of S in the suffix tree of S is the node v such that α is a (not necessarily proper) prefix of $\ell(v)$ and $\ell(u)$ is a proper prefix of α, where u is the parent of v. A *suffix link* connects the node of the suffix tree that corresponds to a string α, to the node of the suffix tree that corresponds to the string $\alpha[2,|\alpha|]$. Inverting the direction of all suffix links gives the so-called *explicit Weiner links*. The *suffix link tree* of S is the trie whose set of nodes consists of the set of all internal nodes of the suffix tree of S, and whose set of edges consists of all the explicit Weiner links (that start from internal nodes) of the suffix tree of S. An internal node of the suffix tree that corresponds to a right-maximal string α is the source of an *implicit Weiner link*, labelled by character c, if string $c\alpha$ occurs in S, but is not right-maximal: the target of such implicit Weiner link is the node that corresponds to the shortest string prefixed by $c\alpha$ that labels a node of the suffix tree. The number of implicit and explicit Weiner links (that start from internal nodes) in the suffix tree of a string $S\$$ of length n is upper-bounded by $2n-2$ [21]. The *generalized suffix tree* of a set of strings $S^1, S^2, \ldots, S^m$ on alphabet $[1,\sigma]$ is the suffix tree of the concatenation $S^1 \cdot \$_1 \cdot S^2 \cdot \$_2 \cdot \cdots \cdot S^m \cdot \$_m$, where $\$_1, \ldots, \$_m$ are distinct separators that are lexicographically smaller than every character in $[1,\sigma]$.

The Burrows-Wheeler transform (BWT) is a standard tool in text indexing. For convenience, we define the Burrows-Wheeler transform only for strings terminated with a unique character $\$ = 0$ that is lexicographically smaller than all characters in Σ. The suffix array $\mathsf{SA}_S[1,n]$ of S is an array such that $\mathsf{SA}_S[i]$ is the starting position of the suffix of S with lexicographic rank i among all suffixes of S. The Burrows-Wheeler transform $\mathsf{BWT}_S[1,n]$ of S is the string such that $\mathsf{BWT}_S[i] = S[\mathsf{SA}_S[i]-1]$ if $\mathsf{SA}_S[i] \neq 1$, and $\mathsf{BWT}_S[i] = S[n]$ otherwise. Given a collection of strings $S^1, S^2, \ldots, S^m$, where $S^i \in \Sigma^+$ for all $i \in [1,m]$, we call *BWT of the collection* the string BWT_S, where $S = S^1 \cdot \$_1 \cdot S^2 \cdot \$_2 \cdot \cdots \cdot S^m \cdot \$_m$, and $\$_1, \ldots, \$_m$ are distinct separators that are lexicographically smaller than every character in Σ. The BWT can be used as a full-text index, by encoding it to answer *rank queries* $\mathtt{rank}_{\mathsf{BWT}}(i,c)$, which return the number of times character $c \in \Sigma$ occurs in the prefix $\mathsf{BWT}[1,i]$, and by augmenting it with array $C[1,\sigma]$, such that $C[i]$ is the number of characters in BWT whose lexicographical rank is strictly less than i. In this paper we assume that the BWT is encoded as a wavelet tree, thus rank operations on the BWT take $O(\log\sigma)$ time [22]. Rank operations on a bitvector of length n take constant time if such bitvector is augmented with suitable data structures of $o(n)$ bits; such data structures can be built in $O(n)$ time and $o(n)$ bits of working space [23, 24]. Rank queries and the C array enable a *backward step* operation on the BWT:

given the lexicographic rank i' of suffix $S[i,n]$, a backward step gives the lexicographical rank of suffix $S[i-1,n]$ using the formula $C[\mathsf{BWT}_S[i']] + \mathtt{rank}_{\mathsf{BWT}}(i', \mathsf{BWT}_S[i'])$. In what follows, we drop the subscript from SA, BWT and rank whenever it is clear from the context.

We can associate to each substring α of S the interval $\mathsf{SA}_S[i,j]$ that contains the starting positions of all the suffixes of S prefixed by α, i.e. the starting positions of all occurrences of α in S. There is a bijection between the set of all such intervals of size at least two and the set of all internal nodes of the suffix tree of S. Given any such interval associated with string α, and given a character c, we can compute the interval of string $c\alpha$ if it exists (or return an empty interval otherwise), using just two rank queries on BWT_S [1]. If α is right-maximal, this operation corresponds to taking a Weiner link labelled by c from the internal node of the suffix tree labelled by α. We can traverse the entire suffix-link tree by performing a linear number of such operations, and by using a suitably designed stack [25]: many algorithms based on the suffix tree can be simulated space-efficiently using such traversal [25].

The *bidirectional Burrows-Wheeler index* [26–29] consists of BWT_S and of $\mathsf{BWT}_{\overline{S}}$, which we also denote by $\overline{\mathsf{BWT}}_S$. BWT can be interpreted as the list of left extensions of all lexicographically sorted suffixes of S, and $\overline{\mathsf{BWT}}$ can be interpreted as the list of right extensions of all *colexicographically* sorted prefixes, where a string α is *colexicographically smaller* than a string β iff $\overline{\alpha}$ is lexicographically smaller than $\overline{\beta}$. A substring α of S is associated with a contiguous lexicographic (respectively, colexicographic) interval, i.e. with the lexicographic (respectively, colexicographic) range of all suffixes (respectively, prefixes) of S that are prefixed (respectively, suffixed) by α. We denote the first and last position of the lexicographic interval of a substring α with $i_\alpha^{\rightarrow}$ and $j_\alpha^{\rightarrow}$, respectively, and the first and last positions of the colexicographic interval of the same substring with $i_\alpha^{\leftarrow}$ and $j_\alpha^{\leftarrow}$, respectively. Given a string α, indices $i_\alpha^{\rightarrow}$, $j_\alpha^{\rightarrow}$, $i_\alpha^{\leftarrow}$, $j_\alpha^{\leftarrow}$ and a character $c \in \Sigma$, it is possible to compute a *left-extension*, i.e. the indices $i_{c\alpha}^{\rightarrow}$, $j_{c\alpha}^{\rightarrow}$, $i_{c\alpha}^{\leftarrow}$, $j_{c\alpha}^{\leftarrow}$ and a *right-extension*, i.e. the indices $i_{\alpha c}^{\rightarrow}$, $j_{\alpha c}^{\rightarrow}$, $i_{\alpha c}^{\leftarrow}$, $j_{\alpha c}^{\leftarrow}$ in time $O(\sigma \log \sigma)$: see Algorithm 1. Within the same space budget, the time complexity can be further improved to $O(\log \sigma)$, by replacing the sum in line 4 of Algorithm 1 with the count operation provided by wavelet trees [30], and finally to $O(1)$ by using monotone minimal perfect hash functions [25]. In what follows, we use extendLeft and extendRight to denote these two primitives of a bidirectional BWT index. We also assume that a bidirectional BWT index provides operation enumerateLeft (respectively, enumerateRight), which, given a string α, $i_\alpha^{\rightarrow}$, $j_\alpha^{\rightarrow}$ (respectively, $i_\alpha^{\leftarrow}$, $j_\alpha^{\leftarrow}$), and a character $c \in \Sigma$, returns the set of all d distinct characters that occur in $\mathsf{BWT}_S[i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}]$ (respectively, in $\overline{\mathsf{BWT}}_S[i_\alpha^{\leftarrow}, j_\alpha^{\leftarrow}]$), in lexicographic order. Operations enumerateLeft and enumerateRight can be implemented in $O(d \log(\sigma/d))$ time using wavelet trees [28].

Algorithm 1 Implementing extendLeft on a bidirectional BWT index. The pseudocode for extendRight is identical, but uses $\overline{\mathsf{BWT}}$ rather than BWT, and swaps $i^{\leftarrow}$ with $i^{\rightarrow}$, and $j^{\leftarrow}$ with $j^{\rightarrow}$.

1: **procedure** EXTENDLEFT(intervals $[i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}]$ and $[i_\alpha^{\leftarrow}, j_\alpha^{\leftarrow}]$, character c)
2: $i_{c\alpha}^{\rightarrow} := C[c] + \mathtt{rank}_{\mathsf{BWT}}(i_\alpha^{\rightarrow}, c)$
3: $j_{c\alpha}^{\rightarrow} := C[c] + \mathtt{rank}_{\mathsf{BWT}}(j_\alpha^{\rightarrow}, c)$
4: $i_{c\alpha}^{\leftarrow} := i_{c\alpha}^{\leftarrow} + \sum_{d \in \Sigma : d < c} \left(\mathtt{rank}_{\mathsf{BWT}}(j_\alpha^{\rightarrow}, d) - \mathtt{rank}_{\mathsf{BWT}}(i_\alpha^{\rightarrow} - 1, d)\right)$
5: $j_{c\alpha}^{\leftarrow} := i_{c\alpha}^{\leftarrow} + (j_{c\alpha}^{\rightarrow} - i_{c\alpha}^{\rightarrow})$
6: **return** $[i_{c\alpha}^{\rightarrow}, j_{c\alpha}^{\rightarrow}]$, $[i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}]$
7: **end procedure**

Methods

We show how to implement in small space the key primitives of the read clustering pipeline, using the bidirectional BWT index of the concatenation of all reads in the sample. Specifically, we focus on the step that builds the connected components, since this is the space bottleneck of the entire pipeline in practice, and since the same techniques can be applied to the initial filtering of reads from low-frequency taxa. Building composition vectors and clustering them requires negligible space compared to the other steps.

We say that the *rank of a read* is the number of reads that come before it in the concatenation, plus one to make the ranks start from one. We first describe how to iterate over all the RC right-maximal substrings of S, a result that will be useful in what follows:

Lemma 1 *Given the bidirectional BWT index of a string $S \in [1,\sigma]^{n-1}\,\$$, where $\$ = 0$, we can iterate over all the RC right-maximal substrings of S in $O(n \log \sigma)$ time and $O(\sigma \log^2 n)$ bits of space in addition to the input and the output.*

Proof We use the recursive procedure in Algorithm 2 to enumerate all the nodes of the generalized suffix tree of S and $\widetilde{S}$, as described in [25]. Each frame in the iteration stack represents the four intervals that identify the lexicographic and colexicographic ranges of a string and its reverse complement. To decide whether substring $c\alpha$ is RC right-maximal, we just need intervals $[i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}]$ and $[i_{\widetilde{c\alpha}}^{\rightarrow}, j_{\widetilde{c\alpha}}^{\rightarrow}]$. Recall that interval $[i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}]$ in the reverse BWT lists all the right extensions of $c\alpha$, and interval $[i_{\widetilde{c\alpha}}^{\rightarrow}, j_{\widetilde{c\alpha}}^{\rightarrow}]$ in the forward BWT lists all the

left extensions of $\widetilde{c\alpha}$. Let $\Sigma'_1 = \{c : c \in \overline{\mathsf{BWT}}[\, i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}\,]\}$ be the set of distinct characters in $\overline{\mathsf{BWT}}[\, i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}]$, and let $\Sigma'_2 = \{\widetilde{c} : c \in \mathsf{BWT}[\, i^{\rightarrow}_{\widetilde{c\alpha}}, j^{\rightarrow}_{\widetilde{c\alpha}}\,]\}$ be the set of distinct *reverse complements* of the characters in $\mathsf{BWT}[\, i^{\rightarrow}_{\widetilde{c\alpha}}, j^{\rightarrow}_{\widetilde{c\alpha}}]$. String $c\alpha$ is RC right-maximal iff $\left|\Sigma'_1 \cup \Sigma'_2\right| > 1$: this can be checked by calling the `enumerateLeft` and `enumerateRight` operations provided by the bidirectional index, and by taking the union of their output.

Every element in the output of `enumerateLeft` and `enumerateRight` can be either charged to an implicit or explicit Weiner link of the generalized suffix tree of S and $\widetilde{S}$, or to an edge of the same tree, thus the total number of such calls is $O(n)$, and the total number of calls to `extendLeft` and `extendRight` is $O(n)$ as well. The claimed time bound comes from properties described in the "Background" section. As used in Algorithm 2, the stack takes $O(\lambda_S \sigma \log n)$ bits, since in the worst case it consists of λ_S levels, each of which contains up to σ quadruplets of intervals in BWT and $\overline{\mathsf{BWT}}$. We reduce the number of levels in the stack to $O(\log n)$ by pushing first, at every iteration, the left-extension with largest sum of interval lengths in, say, BWT, as described in [25]. □

Recall that in our case S is a collection of reads, thus, even without applying the logarithmic stack technique described in [25], the space used by Lemma 1 is $O(\ell\sigma \log n)$ bits, where ℓ is the maximum length of a read.

Algorithm 2 Iterating over all RC right-maximal substrings of a string using the bidirectional BWT index.

```
stack ← Empty iteration stack
push (([1,n],[1,n]),([1,n],[1,n])) to stack
while stack is not empty do
    [i→_α,j→_α],[i←_α,j←_α],[i→_α̃,j→_α̃],[i←_α̃,j←_α̃] ← pop stack
    Σ1 ← enumerateLeft([i→_α,j→_α])
    Σ2 ← enumerateRight([i←_α̃,j←_α̃])
    Σ3 ← {c̃ | c ∈ Σ2}
    for c ∈ Σ1 ∪ Σ3 do
    [i→_cα,j→_cα],[i←_cα,j←_cα] ← extendLeft([i→_α,j→_α],[i←_α,j←_α],c)
    [i→_cα̃,j→_cα̃],[i←_cα̃,j←_cα̃] ← extendRight([i→_α̃,j→_α̃],[i←_α̃,j←_α̃],c̃)
    if cα is RC right-maximal then
        report cα
        stack.push([i→_cα,j→_cα],[i←_cα,j←_cα],[i→_cα̃,j→_cα̃],[i←_cα̃,j←_cα̃])
    end if
    end for
end while
```

We compute k-RC connected components in two steps: first, we compute connected components of the k-relation on reads. Then, we merge every two connected components C_1 and C_2 for which there is a k-mer α such that α is contained in some read in C_1, and $\widetilde{\alpha}$ is contained in some read in C_2. As done in [15], we use a union-find data structure on the set of reads to implement the merging operations (see e.g. [31]). We assume that such data structure supports the following queries: `find`(r), which returns the handle of the connected component containing read r; `union`(C_1, C_2), which merges components C_1 and C_2 and returns the handle of the resulting component; and `size`(C), which returns the number of reads in component C. We initialize the data structure so that every read belongs to a distinct component.

Lemma 2 *Let $S = S^1\$S^2\$\cdots S^m\$\$$ be a string of length n such that $S^i \in \Sigma^+$ for all $i \in [1,m]$, and $\$ = 0$ is a separator. Assume that we are given the bidirectional BWT index of S, a union-find data structure initialized with m sets and supporting* `find` *and* `union` *in time t, and an integer k. Then we can encode, in the union-find data structure, all the connected components of the k-relation graph on set $\{S^1, S^2, \ldots, S^m\}$, in $O(n(t + \log\sigma'))$ time and in $n + o(n) + O(\max\{\ell\sigma' \log n, K \log m\})$ bits of space in addition to the input, where $\sigma' = \sigma + 1$, $\ell = \max\{|S^i| : i \in [1,m]\}$ and K is the number of distinct k-mers of S.*

Proof We enumerate the nodes of the suffix tree of S in the order induced by the suffix-link tree of S, using a recursive procedure similar to Algorithm 2. Specifically, we keep just $[\, i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}\,]$ and $[\, i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]$ for every right-maximal substring α of S, and we use the fact that α is right-maximal iff $\overline{\mathsf{BWT}}[\, i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]$ contains at least two distinct characters (see [25] for further details). Note that the BWT intervals of distinct k-mers are disjoint. Thus, during the iteration, we mark in a bitvector B, of length equal to the size of BWT, the first position of the lexicographic interval of every k-mer. This can be done as follows (see e.g. [21]). We initialize B to all ones and, whenever we enumerate a right-maximal substring α of length at least k, we use operations `enumerateRight` and `extendRight` provided by the bidirectional index to compute the interval $[\, i^{\rightarrow}_{\alpha c}, j^{\rightarrow}_{\alpha c}]$ of every right-extension αc of α, in lexicographic order. Then, we flip bit $B[\, i^{\rightarrow}_{\alpha c}]$ for all c except the first in lexicographic order. At the end of this process we index B to answer rank queries in constant time, so that we can compute the ID of the k-mer whose BWT interval contains a given position i in BWT by $\mathtt{rank}_B(i)$.

Every k-mer interval is associated with the set of distinct reads that contain the starting points of the suffixes of S inside the interval. For each k-mer interval, we store the handle of one of such read. The handles are stored in

an array H, of length equal to the number of k-mer intervals, such that the handle corresponding to the interval of the k-mer that contains position i in BWT is stored in $H[\,\mathtt{rank}_B(i)]$. We initialize array H with null values. Then, we backward-search string S in BWT_S, maintaining the lexicographic rank $i^{\rightarrow}$ of the suffix that starts at the current position, and the rank r of the read that contains such suffix. At each step we compute p, the ID of the k-mer whose interval contains $i^{\rightarrow}$: if $H[\,p]$ is null, we set $H[\,p]$ to $\mathtt{find}(r)$; otherwise, if $H[\,p]$ is different from $\mathtt{find}(r)$, we set $H[\,p]$ to the output of $\mathtt{union}(H[\,p]\,,\mathtt{find}(r))$. □

Note that in Lemma 2 we do not use a distinct separator for every read, but instead we use the same separator for all reads. The result is unaffected by this change, and we will use this convention in the rest of the paper. We leave details to the reader.

Clearly it suffices to consider just k-mers that do not contain \$ and that occur at least twice in S. More tightly, it suffices to consider just right-maximal k-mers that do not contain \$: indeed, if a k-mer α is always followed in S by character c, then the set of reads that are merged by α is a subset of the set of reads that are merged by $\alpha[\,2,k]\cdot c$. Lemma 2 can be adapted to use just right-maximal k-mers:

Corollary 1 *Lemma 2 can be implemented in* $n+o(n)+O(\max\{\ell\sigma'\log n, K'\log m\})$ *bits of space in addition to the input, where* K' *is the number of distinct right-maximal* k*-mers of* S*.*

Proof We follow the same approach as in Lemma 2. The intervals of all right-maximal k-mers are disjoint and of size at least two. We mark the first and the last position of every such interval in array B, by iterating over all right-maximal substrings of S and by setting $B[\,i^{\rightarrow}_{\alpha}]=1$ and $B[j^{\rightarrow}_{\alpha}]=1$ for every right-maximal α of length k. This marking technique was introduced independently by [21, Lemma 16.5] and by [32]. As we backward-search S in BWT_S, we decide whether the k-mer that prefixes the current suffix of S is right-maximal, by checking whether $B[\,i]=1$ or $\mathtt{rank}_B(i)$ is odd, where i is the lexicographic rank of the current suffix. We proceed only in the positive case, using the handle that corresponds to the k-mer located at position $\lceil\mathtt{rank}_B(i)/2\rceil$ of H. □

Note that the running time of Corollary 1 is $O(\mathtt{occ}\cdot t+n\log\sigma)$, where occ is the total number of occurrences of all right-maximal k-mers. In real datasets, for typical values of k (e.g. 36), the number of distinct k-mers can be approximately 45 times bigger than the number of distinct right-maximal k-mers, and the length of the string can be approximately 17 times bigger than the number of occurrences of right-maximal k-mers.

Consider the set $\mathcal{R}$ of all distinct maximal repeats of S of length at least k: every substring $S[\,i,j]$ that equals a right-maximal k-mer of S is a suffix of a substring $S[\,i',j]$, with $i'\le i$, that equals a maximal repeat in $\mathcal{R}$, and every substring $S[\,i',j]$ that equals a maximal repeat in $\mathcal{R}$ has a right-maximal k-mer as a suffix. Thus, issuing union queries using the elements of $\mathcal{R}$ is equivalent to issuing union queries using all right-maximal k-mers. The size of $\mathcal{R}$, however, is *at least* the number of right-maximal k-mers. Specifically, the number of right-maximal k-mers equals the size of set $\mathcal{R}'\subseteq\mathcal{R}$, where $\mathcal{R}'$ is the set of elements of $\mathcal{R}$ that do not have another element of $\mathcal{R}$ as a suffix. In other words, the elements of $\mathcal{R}'$ are the reversed labels of the loci of the reversed right-maximal k-mers of S in the suffix tree *of the reverse of* S.

More tightly, every substring α of a maximal repeat $\beta\in\mathcal{R}$ occurs in S whenever β occurs, and possibly at other positions, therefore the union operations induced by β are a subset of the union operations induced by α, and we can safely disregard β for clustering. We are thus interested in the following subset of the maximal repeats of S:

Definition 3 *Let* $S\in\Sigma^n$ *be a string and let* k *be an integer. A repeat of* S *is called* k*-submaximal if it is a maximal repeat of* S *of length at least* k*, and if it does not contain any maximal repeat of length at least* k *as a substring.*

Note that the set of k-submaximal repeats is a subset of $\mathcal{R}'$. Lemma 2 can be adapted to use just the k-submaximal repeats of S:

Corollary 2 *Lemma 2 can be implemented in* $2n+o(n)+O(\max\{\ell\sigma'\log n, K''\log m\})$ *bits of space in addition to the input, where* K'' *is the number of distinct* k*-submaximal repeats of* S*.*

Proof Since the set of all k-submaximal repeats is a subset of $\mathcal{R}'$, and since the elements of $\mathcal{R}'$ are the reversed labels of the loci of the reversed right-maximal k-mers of S in the suffix tree of $\overline{S}$, there is a one-to-one correspondence between the set of occurrences of k-submaximal repeats and the set of occurrences of their right-maximal suffixes of length k. We can thus issue union queries using just right-maximal k-mers that are the (not necessarily proper) suffix of a k-submaximal repeat, or equivalently using just right-maximal k-mers such that the label of the locus of their reverse in the suffix tree of $\overline{S}$ is the reverse of a k-submaximal repeat.

Assume that we have bitvector B from Corollary 1, with the intervals of all right-maximal k-mers marked with ones, indexed to support rank queries. We mark in another bitvector B' (initialized to all zeros) the subset of such intervals that correspond to k-mers that are the suffix of a k-submaximal repeat, as follows. We scan B

sequentially, and for every pair (i,j) of ones such that the first has odd rank x and the second has even rank $x+1$, we check whether $\mathsf{BWT}_S[\,i,j]$ contains at least two distinct characters: if so, the right-maximal k-mer α that corresponds to interval $[\,i,j]$ is also left-maximal, α is a k-submaximal repeat, and we set $B'[\,i] = B'[\,j] = 1$.

Otherwise, let v be the locus of $\overline{\alpha}$ in the suffix tree of $\overline{S}$, let u be the parent of v, let the label of v be $\ell(v) = \ell(u)\beta\gamma$, let $\overline{\alpha} = \ell(u)\beta$, and let $|\gamma| = g$. We iteratively take backward steps from $[\,i,j]$ until we find a BWT interval that contains at least two distinct characters. This is equivalent to reading the characters of γ sequentially. Let such sequence of backward steps yield intervals $[\,i_1,j_1], [\,i_2,j_2], \ldots, [\,i_g,j_g]$ corresponding to right-maximal strings $\gamma[\,1]\alpha, \gamma[\,2]\gamma[\,1]\alpha, \ldots, \overline{\gamma}\alpha$. Assume that, using rank queries on B, we detect that one such interval $[\,i_y,j_y]$ is contained inside the interval of a right-maximal k-mer θ. Let v' be the locus of $\overline{\theta}$ in the suffix tree of $\overline{S}$. Then $\ell(v')$ is a substring of $\ell(v)$ and $\overline{\ell(v')}$ is an element of $\mathcal{R}'$, thus $\overline{\ell(v)}$ is not k-submaximal, we leave $B'[\,i]$ and $B'[\,j]$ to zero, and we move to the next pair of ones in B. If none of the intervals $[\,i_1,j_1], \ldots, [\,i_g,j_g]$ is contained inside the interval of a right-maximal k-mer, we set $B'[\,i] = B'[\,j] = 1$ and we move to the next pair of ones in B.

At the end of this process, we index B' for rank queries, we replace the indexed B with the indexed B', and we continue as in Corollary 1. The total number of backward steps performed by the algorithm is $O(n)$, since every step visits a distinct right-maximal substring of S. □

Slightly more involved arguments allow one to shave n bits from the space complexity of Corollary 2. The running time of Corollary 2 is $O(\mathtt{occ} \cdot t + n\log\sigma)$, where `occ` is the total number of occurrences of all k-submaximal repeats. In real datasets, for typical values of k (e.g. 36), the number of right-maximal k-mers can be approximately 1.8 times the number of k-submaximal repeats, and the total number of occurrences of right-maximal k-mers can be approximately 1.5 times the number of occurrences of k-submaximal repeats. Once again, we can discard k-submaximal repeats that contain \$.

Before completing the construction of the k-RC connected components, we note that the technique described in Lemma 2 allows one to detect reads whose k-mers occur all less than τ times in the dataset (without considering reverse complements), in $O(n\log\sigma')$ time and in $n + O(\ell\sigma'\log n)$ bits of space in addition to the input and the output. Once all reads from low-frequency species have been detected, it is also possible to derive the BWT of such reads, as well as the BWT of all reads from high-frequency species, directly from BWT_S. We leave such details to the reader.

To complete the pipeline, we just need to merge all pairs of components C_1 and C_2 that share a reverse complemented k-mer. Once again, it suffices to consider just the RC right-maximal k-mers that occur in both S and $\widetilde{S}$:

Lemma 3 *Let $S = S^1\$S^2\$\cdots S^m\$\$$ be a string of length n such that $S^i \in \Sigma^+$ for all $i \in [\,1,m]$, and $\$ = 0$ is a separator. Assume that we are given the bidirectional BWT index of S, a union-find data structure initialized with m sets and supporting* `find` *and* `union` *in time t, and an integer k. Then we can encode, in the union-find data structure, all the connected components of the k-RC-relation on set $\{S^1, S^2, \ldots, S^m\}$, in $O(n(t+\log\sigma'))$ time and in $2n + o(n) + O(\max\{\ell\sigma'\log n, K'''\log m\})$ bits of space in addition to the input, where $\sigma' = \sigma + 1$, $\ell = \max\{|S^i| : i \in [\,1,m]\}$ and K''' is the number of distinct RC right-maximal k-mers of S.*

Proof Let B_2 and B_3 be two bitvectors, of length equal to the length of BWT, initialized to all zeros. We iterate over every RC right-maximal k-mer α using Algorithm 2: if none of the intervals $[\,i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}]$ and $[\,i_{\widetilde{\alpha}}^{\rightarrow}, j_{\widetilde{\alpha}}^{\rightarrow}]$ is empty, then the reads corresponding to interval $[\,i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}]$ should be in the same connected component as the reads corresponding to interval $[\,i_{\widetilde{\alpha}}^{\rightarrow}, j_{\widetilde{\alpha}}^{\rightarrow}]$. Thus, if $i_\alpha^{\rightarrow} \neq j_\alpha^{\rightarrow}$ we set $B_2[\,i_\alpha^{\rightarrow}] = 1$ and $B_2[\,j_\alpha^{\rightarrow}] = 1$, otherwise we set $B_3[\,i_\alpha^{\rightarrow}] = 1$. Similarly, if $i_{\widetilde{\alpha}}^{\rightarrow} \neq j_{\widetilde{\alpha}}^{\rightarrow}$ we set $B_2[\,i_{\widetilde{\alpha}}^{\rightarrow}] = 1$ and $B_2[\,j_{\widetilde{\alpha}}^{\rightarrow}] = 1$, otherwise we set $B_3[\,i_{\widetilde{\alpha}}^{\rightarrow}] = 1$. At the end of this process we index B_2 and B_3 for rank queries, we allocate a vector H_2 of length equal to the number of distinct intervals marked in B_2, and we store in $H_2[\,i]$ the handle of any read that contains the k-mer that corresponds to the i-th marked interval, by backward-searching S in BWT_S as described in Corollary 1. We similarly fill a vector H_3, of length equal to the number of bits marked in B_3. Finally, we use again Algorithm 2 to iterate over every RC right-maximal k-mer α: if none of the intervals $[\,i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}]$ and $[\,i_{\widetilde{\alpha}}^{\rightarrow}, j_{\widetilde{\alpha}}^{\rightarrow}]$ is empty, we issue $\mathtt{union}(h_1, h_2)$, where $h_1 = H_2[\,\lceil \mathtt{rank}_{B_2}(i_\alpha^{\rightarrow})/2\rceil]$ if $i_\alpha^{\rightarrow} \neq j_\alpha^{\rightarrow}$, otherwise $h_1 = H_3[\,\mathtt{rank}_{B_3}(i_\alpha^{\rightarrow})]$. Similarly, $h_2 = H_2[\,\lceil \mathtt{rank}_{B_2}(i_{\widetilde{\alpha}}^{\rightarrow})/2\rceil]$ if $i_{\widetilde{\alpha}}^{\rightarrow} \neq j_{\widetilde{\alpha}}^{\rightarrow}$, otherwise $h_2 = H_3[\,\mathtt{rank}_{B_3}(i_{\widetilde{\alpha}}^{\rightarrow})]$. □

Note that, if the complementation function reverses the alphabet (and DNA complementation does), we can avoid executing Algorithm 2 twice. Indeed, we could just run Algorithm 2 and mark in a bitvector A the interval of α in BWT, and in a bitvector B the interval of $\widetilde{\alpha}$ in $\overline{\mathsf{BWT}}$, for every RC right-maximal k-mer α that occurs both in S and in $\widetilde{S}$. Then, we could allocate two vectors H_a and H_b, of length equal to the number of marked intervals in A and B, and we could store in $H_a[\,i]$ (respectively, in $H_b[\,i]$) the handle of any read that contains the k-mer corresponding to the i-th marked interval in A (respectively, in B). Finally,

we could issue union($H_a[i], H_b[K'' - i + 1]$) for all $i \in [1, K''']$.

Recall that the very last step of the pipeline consists in extracting repeated substrings of length at least $e > h$ from each connected component. Every such string is a substring of a maximal repeat of $S \cdot \widetilde{S}$ of length at least e. If the user has set $e > k$, every such maximal repeat occurs in exactly one connected component: we could thus extract all the (supermaximal) repeats of $S \cdot \widetilde{S}$ of length at least e, in a single traversal of the generalized suffix-link tree of S and $\widetilde{S}$ and within the same budget as the other algorithms (see [25] for details).

Finally, the value of k in the k-RC-relation can be estimated from the dataset itself: specifically, given a range $[k_x, k_y]$ of possible values, one might want to compute the value of k such that the majority of distinct k-mers of S and $\widetilde{S}$ occur at least twice in $S \cdot \widetilde{S}$, i.e. most of such k-mers are not likely to contain sequencing errors. Such k can be computed within the same time and space budget as the algorithms in this paper, using the algorithm described in [33].

In practice the memory used by the enumeration stack is negligible in all algorithms, the peak space usage of the entire pipeline is achieved by Lemma 3 and, assuming that the bidirectional index takes $2n \log \sigma' + o(n \log \sigma')$ bits, such peak is approximately $2n \log \sigma' + (2m + K''') \log m + 2n + o(n \log \sigma')$ bits. The $2m \log m$ bits of space come from the union-find data structure, which stores for each component a pointer to its parent in a tree structure, and the size of the subtree attached to it to maintain balancing. Note that $2m \log m \in O(n \log \sigma)$ if all reads are distinct. Rather than using the union-find data structure for clustering reads, we could use it for clustering distinct k-mers or repeats, and then we could propagate such clustering to reads (as done e.g. in [19]). This could decrease peak memory when clustering the union of a large number of very similar samples.

Results

The purpose of this section is just to show that our algorithms are practical. Our implementation of the bidirectional BWT index is based on C++, on the SDSL library [34], and on the ropebwt2 library [35]. For simplicity we implement Corollary 1 rather than Corollary 2. We use the multithreading support of the C++ 11 standard library to take advantage of multiple cores.

Specifically, since all our algorithms are traversals of a suffix-link tree, we run them on c parallel cores by dividing the BWT into c intervals of similar length and by assigning each interval to a distinct core. This work balancing technique is effective, since the length of the BWT interval of a node v of the suffix tree correlates well in practice with the number of nodes in the subtree of the suffix-link tree rooted at v. We parallelize the backward search of a sequence of reads in its own BWT by dividing the sequence into c blocks of approximately equal length, and by backward-searching each block in parallel.

We observe that the parallel traversal of the suffix-link tree fails to use more than four cores efficiently, thus more advanced work-balancing strategies might be needed: engineering our implementation to exploit a large number of cores is outside the scope of this paper.

The other purpose of this section is to show the potential of our framework, both in terms of clustering quality and in terms of computational resources, by comparing our implementation (called *bwtCluster*) to a sampler of recent, state-of-the-art tools. Specifically, we compare bwtCluster to MetaCluster [15], MBBC [11] and BiMeta [14]. Such comparisons are inherently unfair, for a number of reasons. First, MetaCluster is a highly engineered, parallel version of the read clustering pipeline, extensively tuned over multiple years both in terms of quality and of speed [15, 36–39]. Comparing bwtCluster to MetaCluster should thus penalize bwtCluster in terms of quality, and possibly of speed. Second, BiMeta and MBBC differ from the pipeline we described, BiMeta is single-threaded, and MBBC uses less than two cores on average, thus they could be penalized in terms of speed. Performing an extensive analysis of the clustering results of our framework, and augmenting it with advanced heuristics to make it as accurate as possible, are outside the scope of this paper.

To the best of our knowledge there is no standard dataset for evaluating the performance of unsupervised metagenomic clustering algorithms yet, thus we experiment with the following samples of increasing complexity. First, we build three simple, error-free datasets, to measure how well an algorithm can separate two species that belong to distinct units at different levels in the taxonomy. Such datasets contain exactly two species each, with tenfold coverage and paired-end reads of length 100 base pairs, with no errors[2]. We call the datasets the species level, genus level and family level datasets, respectively. The reference genomes are taken from the NCBI database, and sampled at random locations of the genomes. Second, we replicate the simulated, high-complexity datasets A, B and C described in [15]. Such datasets have realistic error rates, contain up to a hundred species, and have different fractions of low-abundance species. The datasets are created by feeding the reference genomes from NCBI to the Metasim software by [40]. Third, we pick two real samples: a sample from the human gut catalogue described in [41] containing 1.4 billion base pairs, and a sample from a study on the mouse gut described in [42] containing 830 million base pairs [3].

In simulated datasets, we assess the quality of both the k-RC connected components and of the clusters produced by k-means, using the measures described in [15]. Specifically, suppose there are N species in the dataset, and that

an algorithm outputs M clusters. Let R_{ij} be the number of reads in cluster i that are from species j. We call *precision* the ratio between $\sum_{i=1}^{M} \max_j R_{ij}$ and the number of reads in all clusters, and we call *sensitivity* the ratio between $\sum_{j=1}^{N} \max_i R_{ij}$ and the total number of reads in dataset. For brevity we call *preclusters* the k-RC connected components in what follows. We combine the preclusters and the final clusters produced by both rounds of MetaCluster in order to compute precision and sensitivity. We do not measure clustering quality in real samples, since the truth is not known.

We tried to make our tool as close as possible to MetaCluster by implementing many heuristics found in the MetaCluster papers and even by looking at the source code of MetaCluster, and implementing details not present in the Metacluster papers. Specifically, we do not merge a pair of connected components if either of the components has at least 1000 reads, unless one of the components has size less than 100, and before running k-means we filter all connected components containing less than 200 reads. We set the parameters as recommended in [15], namely we set $k = 16$ and $\tau = 4$ for filtering, we set $k = 36$ for clustering, and we use k-mers of length 5 in the composition vectors clustered by k-means. Other MetaCluster heuristics that are not yet implemented in our tool include issuing union queries in increasing order of k-mer frequency, merging two reads if they contain two k-mers at edit distance one from each other, and a few additional heuristics for growing the sizes of the k-RC connected components. Unlike MetaCluster, our tool runs only a single clustering round, but since the number of filtered reads is small, the effect of this is negligible in final the precision and sensitivity.

We ran the four tools for a maximum of 24 hours on each dataset. The results are shown in Table 1.[4] The tools bwtCluster and BiMeta cannot estimate the number of species in a sample, so we gave the true number of species as parameters to all tools, and we set the number of species to 100 for the real samples. MBBC takes in input an initial guess on the number of species. For the species, genus and family level datasets, when MBBC was given the true number of species as the initial guess, it failed and predicted just one species. With the initial guess of 10 the tool predicted the correct number of species for those datasets, and the numbers reported in Table 1 for such datasets are with the initial guess of 10. For the datasets A, B and C, we gave MBBC the true number of species.

Our tool was the only one which was able to process each dataset within 24 hours without returning an error. The peak memory of our tool was between 3.5 to 14.2 smaller then the competing tools. On the species, genus and family-level datasets, as well as on both real datasets, MetaCluster halted with an error, before even running k-means, saying that the number of clusters was too low, due to low coverage. The same error persisted when we tried to run just the second phase, which is designed to cluster low-frequency species. The peak memory usage of bwtCluster was less than 4 bytes per character (Table 1) and it occurred during the construction of the index (Fig. 1), thus it might be further reduced by replacing the BWT construction library. We could also be more careful in keeping in memory just the data structures that are strictly necessary to each step of the pipeline.

On datasets A, B and C, bwtCluster had approximately 94% of the precision of MetaCluster, both in the final clusters and in the preclusters, suggesting that our clusters are approximately as clean as MetaCluster's. The precluster sensitivity of bwtCluster, however, was just approximately 20% of the precluster sensitivity of MetaCluster, suggesting that bwtCluster fragments species into more preclusters than MetaCluster: this could be caused e.g. by the absence of approximate matching and of other advanced merging heuristics implemented in MetaCluster. Both MBBC and BiMeta generally had smaller precision and sensitivity compared to bwtCluster.

In conclusion, every competing tool we considered is either unstable, or it is significantly slower than our implementation, or it uses significantly more memory, and no competitor with a stable implementation achieves higher precision or sensitivity than bwtCluster on a substantial number of datasets. Finally we note that the implementation of MetaCluster requires that all reads in a sample are of equal length, and have length at most 128 base pairs, whereas our tool has no such restriction.

Discussion and conclusions

We described an algorithmic framework for unsupervised read clustering in small space, based on the bidirectional Burrows-Wheeler index of a metagenomic sample. Specifically, we identified a set of core combinatorial primitives and we implemented them in $O(n(t + \log \sigma))$ time using $2n + o(n) + O(\max\{\ell\sigma \log n, K \log m\})$ bits of space in addition to the index and to a union-find data structure on the set of reads, where n is the total number of characters in the sample, m is the number of reads, σ is the total size of the alphabet, t is the query time of the union-find data structure, and K is a measure of the redundancy of the sample, like the number of distinct right-maximal substrings of fixed length k, or the number of distinct submaximal repeats of length at least k. In practice both σ and t are constant, since t can be for example $O(x)$, where x is the value such that the Ackermann function $A(x, x)$ equals m [31]. Our algorithms are practical, and they can exploit multiple cores by a parallel traversal of the suffix-link tree of the sample.

Since our algorithms use a string index as their substrate, one can build such index just once, and run the algorithms multiple times with different settings of the

Table 1 Precision, sensitivity, peak memory usage and wall clock time of the following clustering algorithms: bwtCluster (BWT), MetaCluster (MC), MBBC, and BiMeta (BM)

Dataset	Size	Tool	Precluster		Cluster		Mem. (GB)	Time
			prec.	sens.	prec.	sens.		
Species level	0.1	BWT	0.84	0.18	0.83	0.8	0.3	3.7 m
		MC	×	×	×	×	×	×
		MBBC			0.7	0.7	3.8	7.3 m
		BM			0.52	0.76	2.6	9.1 m
Genus level	0.1	BWT	0.87	0.09	0.87	0.84	0.3	3.7 m
		MC	×	×	×	×	×	×
		MBBC			0.79	0.79	3.9	8.7 m
		BM			0.59	0.59	2.5	9.1 m
Family level	0.1	BWT	0.94	0.12	0.94	0.91	0.3	3.9 m
		MC	×	×	×	×	×	×
		MBBC			0.78	0.78	3.8	4.6 m
		BM			0.65	0.65	2.6	9 m
A	1.7	BWT	0.84	0.01	0.71	0.34	6.4	1 h
		MC	0.90	0.05	0.76	0.71	22.4	41 m
		MBBC					≥64	≥24 h
		BiMeta					≥69	≥24 h
B	0.6	BWT	0.92	0.01	0.76	0.72	1.9	27 m
		MC	0.97	0.05	0.82	0.37	10.5	11 m
		MBBC					≥27	≥24 h
		BM			0.30	0.52	22	4.6 h
C	1.7	BWT	0.84	0.01	0.69	0.40	5.7	1 h
		MC	0.90	0.05	0.71	0.70	22.3	43 m
		MBBC					≥65	≥24 h
		BM					≥59	≥24 h
Human gut	1.4	BWT					4.1	53 m
		MC					×	×
		MBBC					≥35	≥24 h
		BM					≥17	≥24 h
Mouse gut	0.8	BWT					2.2	25 m
		MC					×	×
		MBBC					≥22	≥24 h
		BM					≥35	≥24 h

The size of each dataset is given in billion base pairs (Gbp). Algorithms that return an error are marked with symbol ×

parameters. Approximately half of the time taken by our implementation is spent in building the index (Fig. 1), thus building the index just once is likely to speed up this frequent use case in explorative data analysis. Since the index is based on the ubiquitous Burrows-Wheeler transform, such transform might have already been computed for supporting other queries, making such algorithms immediately applicable to existing datasets.

Compressed representations of the BWT could reduce peak space even further. Specifically, the BWT of the union of similar metagenomic samples is likely to be very compressible, and since the space used by our algorithms

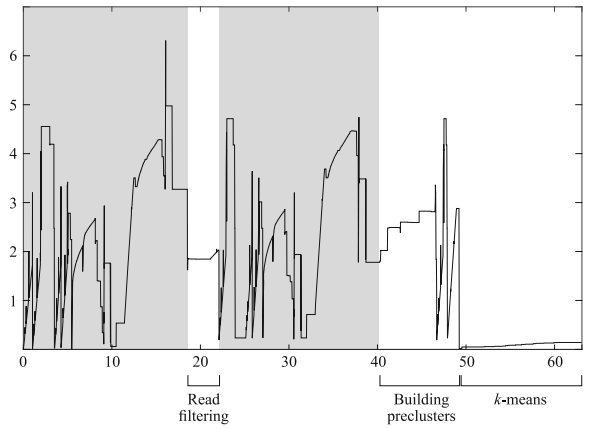

Fig. 1 Running bwtCluster on dataset A: time (*horizontal axis*, minutes) versus memory (*vertical axis*, gigabytes). BWT index constructions are highlighted in *gray*

in addition to the BWT is dominated by a measure of the redundancy of the input, such space is not likely to grow significantly when multiple similar samples are clustered at the same time.

Finally, one could experiment with dropping the k-RC-relation altogether, and with merging reads using just the k-relation: a connected component would then correspond to a substring of a genome *in a specific orientation*, and two connected components that originate from reading the same substring in different orientations would likely be merged during the final k-means step, since their composition vectors are similar. This would remove the need for storing $\overline{\mathsf{BWT}}_S$ in all steps after the initial filtering of reads from low-frequency taxa, since the corresponding algorithms can be implemented on top of the unidirectional traversal described in [43].

Endnotes

[1] More precisely if the interval of α is $[i,j]$ then the interval of $c\alpha$ will be $[i',j']$, where $i' = C[c] + \mathtt{rank}_{\mathsf{BWT}}(i-1,c) + 1$ and $j' = C[c] + \mathtt{rank}_{\mathsf{BWT}}(j,c)$.

[2] The first dataset contains species *Vibrio cholerae* and *Vibrio vulnificus*, the second *Vibrio cholerae* and *Photobacterium gaetbulicola*, the third *Vibrio cholerae* and *Escherichia coli*

[3] EBI identifier SAMEA728599, MG-RAST identifier 4517724.3

[4] We run the species, genus, and family level datasets on a machine with a quad core Intel Core i7-6700K 4 GHz processor and 16GB of DDR4 RAM clocked at 2666 MHz. We run all other datasets on a machine with 1.5 TB of RAM and four Intel Xeon CPU E7-4830 v3 processors (48 total cores, 2.10 GHz each).

Acknowledgements
We thank the anonymous reviewers of a previous submission for helping us improve the presentation, and for pointing us to references [2, 3].

Declarations
This article has been published as part of *BMC Bioinformatics* Volume 18 Supplement 3, 2017. Selected articles from the 15th Asia Pacific Bioinformatics Conference (APBC 2017): bioinformatics. The full contents of the supplement are available online https://bmcbioinformatics.biomedcentral.com/articles/supplements/volume-18-supplement-3.

Funding
This work was supported in part by the Academy of Finland, grant 284598.

Availability of data and materials
Our implementation of the read clustering pipeline is available at [44], and our implementation of the bidirectional BWT index is available at [45]. All source code is available under the GPLv3 license. The real datasets analyzed in this study are described in [41, 42]. The artificial datasets generated in this study are available from the corresponding author on request.

Authors' contributions
All authors designed the algorithms, read and approved the final manuscript. JA implemented the algorithms and performed the experiments. JA and FC drafted the manuscript and carried out the literature study.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] Department of Computer Science, University of Helsinki, Gustaf Hällströmin katu 2b, 00560 Helsinki, Finland. [2] Max Planck Institute for Molecular Cell Biology and Genetics, Pfotenhauerstr. 108, 01307 Dresden, Germany. [3] DTISI, CERIST (Research Centre for Scientific and Technical Information), Rue des 3 Fréres Aissou, 16306 Algiers, Algeria.

Published: 14 March 2017

References

1. Peng Y, Leung HC, Yiu S-M, Chin FY. Meta-IDBA: a de novo assembler for metagenomic data. Bioinformatics. 2011;27(13):94–101.
2. Pell J, Hintze A, Canino-Koning R, Howe A, Tiedje JM, Brown CT. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. Proc Natl Acad Sci. 2012;109(33):13272–13277.
3. Howe AC, Jansson JK, Malfatti SA, Tringe SG, Tiedje JM, Brown CT. Tackling soil diversity with the assembly of large, complex metagenomes. Proc Natl Acad Sci. 2014;111(13):4904–909.
4. Meyer F, Paarmann D, D'Souza M, Olson R, Glass E, Kubal M, Paczian T, Rodriguez A, Stevens R, Wilke A, Wilkening J, Edwards R. The metagenomics RAST server – a public resource for the automatic phylogenetic and functional analysis of metagenomes. BMC Bioinforma. 2008;9(1):1–8.
5. Leinonen R, Sugawara H, Shumway M. The sequence read archive. Nucleic Acids Res. 2010;39(supplement 1):D19–D21. http://nar.oxfordjournals.org/content/39/suppl_1/D19.
6. Su C-H, Wang T-Y, Hsu M-T, Weng FC-H, Kao C-Y, Wang D, Tsai H-K. The impact of normalization and phylogenetic information on estimating the distance for metagenomes. IEEE/ACM Trans Comput Biology Bioinforma. 2012;9(2):619–28.
7. Jiang B, Song K, Ren J, Deng M, Sun F, Zhang X. Comparison of metagenomic samples using sequence signatures. BMC Genomics. 2012;13(1):1.
8. Wang Y, Liu L, Chen L, Chen T, Sun F. Comparison of metatranscriptomic samples based on k-tuple frequencies. PloS ONE. 2014;9(1):84348.
9. Maillet N, Lemaitre C, Chikhi R, Lavenier D, Peterlongo P. Compareads: comparing huge metagenomic experiments. BMC Bioinformatics. 2012;13(19):1.
10. Kelley DR, Salzberg SL. Clustering metagenomic sequences with interpolated Markov models. BMC Bioinformatics. 2010;11(1):544.
11. Wang Y, Hu H, Li X. MBBC: an efficient approach for metagenomic binning based on clustering. BMC Bioinformatics. 2015;16(1):36.

12. Kislyuk A, Bhatnagar S, Dushoff J, Weitz JS. Unsupervised statistical clustering of environmental shotgun sequences. BMC Bioinformatics. 2009;10(1):316.
13. Wu Y-W, Ye Y. A novel abundance-based algorithm for binning metagenomic sequences using ℓ-tuples. J Comput Biol. 2011;18(3):523–34.
14. Van Lang T, Van Hoai T, et al. A two-phase binning algorithm using ℓ-mer frequency on groups of non-overlapping reads. Algorithm Mol Biol. 2015;10(1):1.
15. Wang Y, Leung HC, Yiu S-M, Chin FY. MetaCluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample. Bioinformatics. 2012;28(18):356–62.
16. Siegel K, Altenburger K, Hon Y-S, Lin J, Yu C. Puzzlecluster: A novel unsupervised clustering algorithm for binning dna fragments in metagenomics. Current Bioinformatics. 2015;10(2):231–52.
17. Baran Y, Halperin E. Joint analysis of multiple metagenomic samples. PLoS Comput Biol. 2012;8(2):1–11.
18. Alneberg J, Bjarnason BS, de Bruijn I, Schirmer M, Quick J, Ijaz UZ, Lahti L, Loman NJ, Andersson AF, Quince C. Binning metagenomic contigs by coverage and composition. Nature methods. 2014;11(11):1144–1146.
19. Tanaseichuk O, Borneman J, Jiang T. Separating metagenomic short reads into genomes via clustering. Algorithms Mol Biol. 2012;7(1):1.
20. Weiner P. Linear pattern matching algorithms. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory. Washington, DC, USA: IEEE; 1973. p. 1–11.
21. Mäkinen V, Belazzougui D, Cunial F, Tomescu AI. Genome-Scale Algorithm Design. Cambridge: Cambridge University Press; 2015. ISBN-13: 9781107078536.
22. Grossi R, Gupta A, Vitter JS. High-order entropy-compressed text indexes. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. Baltimore: Society for Industrial and Applied Mathematics Address of symposium; 2003. p. 841–50.
23. Clark D. Compact pat trees. Canada: PhD thesis, University of Waterloo; 1996.
24. Munro I. Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS v. 1180. Hyderabad: Springer; 1996. p. 37–42.
25. Belazzougui D, Cunial F, Kärkkäinen J, Mäkinen V. Versatile succinct representations of the bidirectional burrows-wheeler transform. In: European Symposium on Algorithms. Sophia Antipolis: Springer; 2013. p. 133–44.
26. Lam TW, Li R, Tam A, Wong S, Wu E, Yiu S-M. High throughput short read alignment via bi-directional BWT. In: IEEE International Conference on Bioinformatics and Biomedicine, 2009. Washington D.C: IEEE; 2009. p. 31–6.
27. Li R, Yu C, Li Y, Lam TW, Yiu S-M, Kristiansen K, Wang J. Soap2: An improved ultrafast tool for short read alignment. Bioinformatics. 2009;25(15):1966–1967.
28. Schnattinger T, Ohlebusch E, Gog S. Bidirectional search in a string with wavelet trees. In: 21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010). Lecture Notes in Computer Science. New York: Springer; 2010. p. 40–50.
29. Schnattinger T, Ohlebusch E, Gog S. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inform Comput. 2012;213:13–22.
30. Navarro G. Wavelet trees for all. J Discret Algorithms. 2014;25:2–20.
31. Cormen TH. Introduction to Algorithms. Cambridge, MA, USA: MIT press; 2009.
32. Beller T, Ohlebusch E. Efficient construction of a compressed de Bruijn graph for pan-genome analysis. In: Combinatorial Pattern Matching, Proceedings. Ischia Island: Springer; 2015. p. 40–51.
33. Belazzougui D, Cunial F. A framework for space-efficient string kernels. In: Combinatorial Pattern Matching, Proceedings. Ischia Island: Springer; 2015. p. 13–25.
34. Gog S, Beller T, Moffat A, Petri M. From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms. Copenhagen; 2014. p. 326–37.
35. Li H. Fast construction of FM-index for long sequence reads. Bioinformatics. 2014;30(22):3274–5.
36. Yang B, Peng Y, Leung HC, Yiu S-M, Chen J-C, Chin FY. Unsupervised binning of environmental genomic fragments based on an error robust selection of ℓ-mers. BMC Bioinformatics. 2010;11(Suppl 2):5.
37. Yang B, Peng Y, Leung H, Yiu S-M, Qin J, Li R, Chin FY. MetaCluster: unsupervised binning of environmental genomic fragments and taxonomic annotation. In: Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology; 2010. p. 170–9.
38. Leung HC, Yiu S-M, Yang B, Peng Y, Wang Y, Liu Z, Chen J, Qin J, Li R, Chin FY. A robust and accurate binning algorithm for metagenomic sequences with arbitrary species abundance ratio. Bioinformatics. 2011;27(11):1489–1495.
39. Wang Y, Leung HC, Yiu S-M, Chin FY. MetaCluster 4.0: a novel binning algorithm for NGS reads and huge number of species. J Comput Biol. 2012;19(2):241–9.
40. Richter DC, Ott F, Auch AF, Schmid R, Huson DH. MetaSim – a sequencing simulator for genomics and metagenomics. PloS ONE. 2008;3(10):3373.
41. Qin J, Li R, Raes J, Arumugam M, Burgdorf KS, Manichanh C, Nielsen T, Pons N, Levenez F, Yamada T, et al. A human gut microbial gene catalogue established by metagenomic sequencing. Nature. 2010;464(7285):59–65.
42. Langille MG, Meehan CJ, Koenig JE, Dhanani AS, Rose RA, Howlett SE, Beiko RG. Microbial shifts in the aging mouse gut. Microbiome. 2014;2(1):1.
43. Belazzougui D. Linear time construction of compressed text indices in compact space. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing. New York: ACM; 2014. p. 148–93.
44. Alanko J. bwtCluster: Space-efficient clustering of metagenomic reads using the bidirectional Burrows-Wheeler transform. 2016. https://github.com/jnalanko/bwtCluster. Accessed 06 Oct 2016.
45. Alanko J. BD_BWT_index: Bidirectional BWT text index for byte alphabets. 2016. https://github.com/jnalanko/BD_BWT_index. Accessed 06 Oct 2016.