Space-Efficient Representation of Genomic k-Mer Count Tables

Yoshihiro Shibuya
LIGM, CNRS, Univ. Gustave Eiffel, Marne-la-Vallée, France

Djamal Belazzougui
CAPA, DTISI, Centre de Recherche sur l'Information Scientifique et Technique, Algiers, Algeria

Gregory Kucherov

LIGM, CNRS, Univ. Gustave Eiffel, Marne-la-Vallée, France
Skolkovo Institute of Science and Technology, Moscow, Russia

Abstract

Motivation. k-mer counting is a common task in bioinformatic pipelines, with many dedicated tools available. Output formats could rely on quotienting to reduce the space of k-mers in hash tables, however counts are not usually stored in space-efficient formats. Overall, k-mer count tables for genomic data take a considerable space, easily reaching tens of GB. Furthermore, such tables do not support efficient random-access queries in general.

Results. In this work, we design an efficient representation of k-mer count tables supporting fast random-access queries. We propose to apply Compressed Static Functions (CSFs), with space proportional to the empirical zero-order entropy of the counts. For very skewed distributions, like those of k-mer counts in whole genomes, the only currently available implementation of CSFs does not provide a compact enough representation. By adding a Bloom Filter to a CSF we obtain a Bloom-enhanced CSF (BCSF) effectively overcoming this limitation. Furthermore, by combining BCSFs with minimizer-based bucketing of k-mers, we build even smaller representations breaking the empirical entropy lower bound, for large enough k. We also extend these representations to the approximate case, gaining additional space. We experimentally validate these techniques on k-mer count tables of whole genomes (*E.Coli* and *C.Elegans*) as well as on k-mer document frequency tables for 29 *E.Coli* genomes. In the case of exact counts, our representation takes about a half of the space of the empirical entropy, for large enough k's.

2012 ACM Subject Classification Applied computing

Keywords and phrases k-mer counting, data structures, compression, minimizers, compressed static function, Bloom filter, empirical entropy

Digital Object Identifier 10.4230/LIPIcs.WABI.2021.8

Supplementary Material *Software (Source Code)*: `https://github.com/yhhshb/locom.git`

Funding *Gregory Kucherov*: partially funded by RFBR project 20-07-00652 and joint RFBR and JSPS project 20-51-50007.

1 Introduction

Nowadays, many bioinformatics pipelines rely on k-mers to perform a multitude of different tasks. Representing sequences as sets of words of length k generally leads to more time-efficient algorithms than relying on traditional alignments. For these reasons, alignment-free algorithms have started to replace their alignment-based counterparts in a wide range of practical applications, from sequence comparison and phylogenetic reconstruction [34, 36,

© Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov;
licensed under Creative Commons License CC-BY 4.0

7, 10] to finding SNPs [28, 15] and other tasks. These algorithms often require to associate some kind of information to k-mers involved in the analysis, that is, to build maps where keys are k-mers.

Typical values to associate to k-mers are their frequencies in a particular dataset. Actual counting can be performed by one of several available k-mer counting tools developed in recent years [16, 29, 24, 33]. Counting tables generally include both k-mers and counts requiring considerable amounts of disk space to be stored. For example, the KMC output for a human genome with $k = 32$ weights in at around 28GB.

In many scenarios, it is useful to store k-mer count tables independently from the sequence data in order to retrieve counts multiple times and avoid redundant heavy counting steps. The original sequence dataset can be used as the primary source of k-mers while a random-access data structure will then allow retrieving their counts efficiently. Among potential applications of such a data structure, efficient representation of k-mer counts can be useful for read correction [21]. More generally, information about k-mer counts is increasingly used in other applications [34, 28, 15, 23, 13, 14, 26] which can benefit from space-efficient solutions.

In many applications, space can be significantly reduced by representing the mapping without actually storing k-mers. Minimal Perfect Hash Functions (MPHFs for short) implement such an approach [27, 37, 9]. A MPHF bijectively maps each item from a set S to an index in the range $[0, |S| - 1]$. Any additional information can then be stored in an array indexed by the values returned by the MPHF. Practical applications of MPHFs in bioinformatics include [22] and [38].

Frequency distributions of k-mer counts in genomic data are low-entropy distributions, unless k is small. It is in fact known that k-mer counts for genomic sequences follow a heavy-tail distribution [6, 4]. For large enough k-mer lengths, counts tend to follow a skewed power-law distribution with the majority of k-mers occurring only few times, mostly once. For these reasons, the multiset of counts of k-mers will typically have a fairly low empirical zero-order entropy and therefore could be effectively compressed to save further space. However, simply compressing the count array does not maintain queryability, which requires specialized algorithms for this task. Note also that MPHFs themselves encompass a non-negligible space overhead, with the theoretical minimum of 1.44 bits/key. BBHash [22], a popular MPHF implementation for bioinformatics, requires around 3 bits per element in addition to the space for storing the values.

Maps on static sets of keys can be encoded using so-called Static Functions [1, 11]. Unlike MPHFs, the actual hash function and the values are encoded into the same structure. Compressed Static Functions (CSFs) try to benefit from the compressibility of the value array and approach the number of bits defined by the empirical entropy. This feature makes them particularly useful for representing different k-mer annotations, such as counts or presence information across sequences of a given sample [23, 13, 14, 26]. CSFs can be used as readily available drop-in replacements of MPHFs whenever the set of queryable k-mers is known in advance. Solutions based on MPFHs and Static Functions assume that only k-mers present in the datasets can be queried for their frequency. In many cases, this is not restrictive as the "universe" of query k-mers can be effectively specified: for example, it can be restricted to k-mers from a given genome or a pan-genome. It is also conceivable to add an appropriate structure providing presence-absence information, in order to benefit from the reduction of space provided by a compact count representation.

The goal of this paper is to study data structures for storing genomic k-mer count tables using the smallest possible space. For this, we combine different tools: CSFs and Bloom filters on the one hand, and minimizers on the other hand.

Our first contribution is the enhancement of CSFs with a Bloom Filter to deal with datasets of very small entropy and to achieve better space usage. We chose an *E.Coli* and a *C.Elegans* genome to test our implementation. With these examples, we demonstrate the advantages of our BCSF implementation over a simple CSF.

Our second improvement takes advantage of the fact that similar k-mers tend to have identical (or similar) counts (see also [23]). Following this insight, we introduce a minimizer-based bucketing scheme to cluster together count values of k-mers with the same minimizer. A similar idea is used by some k-mer counting algorithms [29, 16, 18] with the difference that in our case buckets contain counts rather than k-mers themselves. By choosing a representative value for each bucket, we obtain a "bucket table" that we encode using Bloom-enhanced CSF. Bucketing allows us to break the empirical entropy lower bound, as we show on both *E.Coli* and *C.Elegans* examples. To demonstrate the advantages of bucketing on higher-entropy distributions, we apply it to represent a table of "document frequencies" [12, 5, 25, 17] of each k-mer across multiple strains of *E.Coli*.

We study different implementation schemes based on these ideas and compare their space performance, as well as associated query time. For large enough k (and large enough minimizers lengths), we are able to consistently break the barrier of the empirical entropy of the input table. To the best of our knowledge, this is the first implementation proposing such a compact representation. We also study an extension to the approximate case when query answers are within a pre-defined absolute error from the true value, for which we achieve an even smaller space.

2 Technical preliminaries

Throughout the paper we consider a k-mer count table to be an associative array f mapping a set of k-mers K, considered static, to their counts, i.e. number of occurrences in a given dataset. $||f||_1$ stands for the L1-norm of f, that is $\sum_{q \in K} f(q)$.

2.1 Minimizers

Minimizers are a popular technique used in different applications involving k-mer analysis. Given a k-mer q of length k, its minimizer of length m, with $m \leq k$, is the smallest substring of q of length m w.r.t. some order defined on m-mers. The use of minimizers for biosequence analysis goes back to [30], whereas a similar concept, named *winnowing*, has been earlier applied in [32] to document search. The guiding idea is that a minimizer can be considered as a "footprint" (hash value) of a corresponding k-mer so that similar (e.g. neighboring in the genome) k-mers are likely to have the same minimizer. Thus, if the order of m-mers is randomly chosen, minimizers can be seen as a specific instance of locality-sensitive hashing, in particular of MinHash sketching [3].

Minimizers have been successfully applied to various data-intensive sequence analysis problems in bioinformatics, such as metagenomics (Kraken [35]) or minimizing cache misses in k-mer counting (KMC [16]), or mapping and assembling long single-molecule reads [19, 20]. Recently, there has been a series of works on both theoretical and practical aspects of designing efficient minimizers, see e.g. [39, 8] and references therein.

2.2 Bloom filters

Bloom filter is a very common probabilistic data structure that supports membership queries for a given set S drawn from a large universe U, admitting a controlled fraction of *false positives*. To insure a false positive rate ε, that is the probability ε for an item from $U \setminus S$ to

be erroneously classified as belonging to S, a Bloom filter B requires $|S| \log e \log \frac{1}{\varepsilon}$ bits, i.e. $\approx 1.44 \log \frac{1}{\varepsilon}$ bits per element of S. For a set $T \subseteq U \setminus S$, we denote $FP_B(T)$ the set of false positives of T, of expected size $\varepsilon|T|$.

2.3 Compressed static functions

A static function (SF) is a representation of a function defined on a given subset S of a universe U such that an invocation of the function on any element from S yields the function value, while an invocation on an element from $U \setminus S$ produces an arbitrary output. The problem has been studied in several works (see references in [1, 11]) resulting in several solutions that allow function values to be retrieved without storing elements of S themselves. One natural solution comes through MPHFs: one can build an MPHF for S and then store function values in order in a separate array. This solution, however, incurs an overhead associated with the MPHF, known to be theoretically lower-bounded by about 1.44 bits per element of S.

This overhead is especially unfortunate when the distribution of values is very skewed, in which case the value array may be compressed into a much smaller space. Compressed Static Functions try to solve this problem by proposing a static function representation whose size depends on the *compressed* value array. The latter is usually estimated through the zero-order empirical entropy, defined by $H_0(f) = \sum_{\ell \in L} \frac{|f^{-1}(\ell)|}{|K|} \log(\frac{|K|}{|f^{-1}(\ell)|})$, where L is the set of all values (i.e. $L = \{f(t) \,|\, t \in K\}$) and $f^{-1}(\ell) = \{t \,|\, f(t) = \ell\}$ is the set of k-mers with count ℓ. $|K| \cdot H_0(f)$ can be viewed as a lower bound on the size of compressed value array, in absence of additional assumptions. Thus, the goal of CSFs is to approach the bound of $H_0(f)$ bits per element as closely as possible, in representing a static function f.

An overview of different algorithmic solutions for SFs and CSFs is out of scope of this paper, we refer the reader to [1, 11] and references therein. [1] proposed a solution for CSF taking an asymptotically optimal $nH_0(f) + o(nH_0(f))$ space (n size of the underlying value set), however the solution is rather complex and probably not suitable for practical implementation. As of today, to our knowledge, the only practical implementation of a CSF is GV3CompressedFunction [11], found in the Java package Sux4J (`https://sux.di.unimi.it/`). Although entropy-sensitive, the method of [11], however, has an intrinsic limitation of using at least 1 bit per element, due to involved coding schemes. This is a serious limitation when dealing with very skewed distributions of values, where one value occurs predominantly often and the empirical entropy can be much smaller than 1. This is precisely the case for count distributions in whole genomes, studied in this paper.

3 Representation of low-entropy data

As mentioned earlier, Compressed Static Functions (CSF) of [11] do not properly deal with datasets generated by low-entropy distributions, in particular with entropy smaller than 1. This case occurs when the dataset has a dominant value representing a large fraction (say, more than a half) of all values. This is typically the case with genomic k-mer count data, especially whole-genome data, where a very large fraction of k-mers occur just once. For example, in *E.Coli* genome (≈5.5Mbp), about 97% of all distinct 15-mers occur once, and only the remaining 3% of 15-mers occur more than once.

For such datasets, the method of [11] does not approximate well the empirical entropy, as it cannot achieve less than 1 bit per key. Here we propose a technique to circumvent this deficiency in order to achieve, in combination with CSFs of [11], a compression close to the empirical entropy.

We build a Bloom filter for all k-mers whose value is not the dominant one, and then construct a CSF on all positives (i.e. true and false positives) of this filter. At query time, we first check the query k-mer against the Bloom filter and, if the answer is positive, recover its value with the CSF.

Formally, let K_0 be the k-mers with the most common frequency. Let $|K_0| = \alpha|K|$. Assume that our Bloom filter implementation takes $C_{BF} \log \frac{1}{\varepsilon}$ bits per key (a standard value is $C_{BF} = \log e \approx 1.44$) and our CSF implementation takes C_{CSF} bits per key. As explained earlier, C_{CSF} depends on the data, however, for the purpose of this section, we abstract from this dependency. In Sec. 6 below, we will specify C_{CSF} for the implementation we use.

We store keys $K \setminus K_0$ in a Bloom filter B and build a CSF for $(K \setminus K_0) \cup FP_B(K_0)$. The total space is

$$C_{BF}(1-\alpha)|K| \log \frac{1}{\varepsilon} + C_{CSF}|K|((1-\alpha) + \varepsilon\alpha). \tag{1}$$

The Bloom filter enables space saving only if α is sufficiently large. To decide if we need a Bloom filter, we have to verify if the inequality

$$C_{BF}(1-\alpha)|K| \log \frac{1}{\varepsilon} + C_{CSF}|K|((1-\alpha) + \varepsilon\alpha) < C_{CSF}|K|. \tag{2}$$

holds for some $\varepsilon < 1$. Note again that C_{CSF} on the left and right sides are not exactly the same in reality, however assuming them the same is not reductive because of specificities of the CSF implementation we use. We will elaborate further on this later on. Then (2) rewrites to

$$\frac{C_{BF}}{C_{CSF}} \frac{1-\alpha}{\alpha} \log \frac{1}{\varepsilon} + \varepsilon < 1. \tag{3}$$

Using simple calculus, we obtain that if $\frac{C_{BF}}{C_{CSF}} \frac{1-\alpha}{\alpha} > \ln 2$ (that is, $\frac{C_{BF}}{C_{CSF}} \frac{1-\alpha}{\alpha} \log e > 1$), then (3) never holds for $0 < \varepsilon < 1$. The left-hand side of (3) reaches its minimum for

$$\varepsilon_0 = \frac{C_{BF}}{C_{CSF}} \frac{1-\alpha}{\alpha} \log e, \tag{4}$$

and this minimum is smaller than 1 if $\varepsilon_0 < 1$. We conclude that in order to decide if a Bloom filter enables space saving, we have to check the value ε_0. If $\varepsilon_0 \geq 1$, we do not need a Bloom filter, otherwise we need one with $\varepsilon = \varepsilon_0$. This shows that a Bloom filter is needed whenever

$$\alpha > \frac{C_{BF} \log e}{C_{CSF} + C_{BF} \log e} \tag{5}$$

For $C_{BF} = C_{CSF}$, this gives $\alpha > 0.59$.

In the rest of the paper we use the term *Bloom-enhanced Compressed Static Function*, BCSF for short, to speak about CSF possibly augmented by a prior Bloom filter, as described in this section. Algorithm 1 summarizes the computation of the BCSF data structure.

4 Using minimizers

4.1 Bucketing

A key idea to reduce the computational burden of counting k-mers, is to use minimizers to bucket k-mers and split the counting process across multiple tables (cf e.g. [16]). Here we use the same principle to bucket count values instead of k-mers themselves. Let $M_m(K) =$

Algorithm 1 BCSF construction.

Data: A counting table f between keys and (integer) values
Result: A BCSF for f
Compute R, the k-mer spectrum of f;
Compute ε with (4);
if $\varepsilon < 1$ **then**
  let $K_0 \subseteq K$ be the k-mers with the most common item in R;
  $C = K \setminus K_0$;
  Initialise Bloom filter B of $\lceil \log(e)|C| \log_2(\frac{1}{\varepsilon})/\rceil$ bits;
  Insert C into Bloom filter;
  Compute $E = FP_B(K_0)$;
  $S = C \cup E$;
else
  $S = K$
end
Construct CSF for S;

$\{\mu_m(q) \mid q \in K\}$ be the set of minimizers of all k-mers of K of a given length $m < k$. We map the input set K onto the (smaller) set $M_m(K)$. To each minimizer $s \in M_m(K)$, corresponds the bucket $\{f(q) \mid q \in K, \mu_m(q) = s\}$. We call a minimizer and the corresponding bucket *ambiguous* if this set contains more than one value. The guiding idea is to replace f by a mapping g of $M_m(K)$ to $\mathbf{N}$. Querying value $f(q)$ for a k-mer $q \in K$ will reduce to first querying $g(\mu_m(q))$ and then possibly "correcting" the retrieved value. In other words, for each bucket, we replace its set of counts with one representative value and we split the query into two operations: retrieving the representative from the buckets and correcting to reconstruct the original value. The rationale is that k-mers having the same minimizer tend to have the same count allowing multiple values to be dealt with by a single bucket.

We consider two implementations which differ on how the representatives are chosen and how corrections are applied. In the *first implementation*, that we name FIL (from FILtration, see Algorithm 2), $g(s)$ is defined to be the majority value among all values of its bucket, ties resolved arbitrarily. In particular, if s is a non-ambiguous minimizer then $g(s)$ is set to the unique value of the bucket. In practice, computing the majority value may incur a computational overhead as this requires storing bucket values until all values are known. An option to cope with this is to use the "approximate majority" computed by the online Boyer-Moore majority algorithm [2].

We then store a "correcting mapping" $h : K \to \mathbf{N}$ defined by $h(q) = f(q) - g(\mu_m(q))$. That is, we construct another counting table h where each k-mer is associated to the correction factor $h(q)$, which, added to the representative $g(s)$ results in the original count c. Both mappings g and h are stored using BCSFs.

The rationale for this scheme is that, due to properties of minimizers, $h(q)$ is supposed to be often 0, which makes h well compressible using BCSF. Note that because of the majority rule, 0 will always be the majority value of h. Therefore, the Bloom filter of the BCSF storing h (if any) will hold k-mers q with $f(q) \neq g(\mu_m(q))$ (i.e. $h(q) \neq 0$). Then the CSF will store h restricted to k-mers with $h(q) \neq 0$ together with a subset of k-mers (false positives of the Bloom filter) for which $h(q) = 0$.

In our *second implementation*, named AMB (from AMBiguity, see Algorithm 3), for non-ambiguous minimizers u, $g(u)$ is again defined to be the unique value of the bucket. For ambiguous minimizers v, we set $g(v) = 0$, where 0 is viewed as a special value marking

Algorithm 2 FIL construction algorithm.

```
Data: A mapping f of keys to (integer) values, a minimizer length m_0
Result: FIL compressed structure
Sort L by increasing order;
T = f;
Initialise an array A of buckets;
for (q, c) in f do
  | z = μ_{m_0}(q) Insert c into g(z);
end
for b in A do
  | Select representative r of bucket b by majority rule;
end
Compress A by using BCSF;
Create output table O;
for (q, c) in f do
  | if g(μ_m(q)) ≠ c then
  |   | Write q and c − g(μ_m(q)) to O;
  | end
end
Compress O by using BCSF;
```

ambiguous buckets (k-mers with count 0 are not present in the input). This has the disadvantage of providing no information about the values of ambiguous buckets, and also of making g less compressible (because of an additional value). On the other hand, this has the advantage of distinguishing between ambiguous and non-ambiguous buckets and allows the query to immediately return the answer for k-mers hashing to non-ambiguous buckets. As a consequence, unambiguous k-mers are not propagated to the second layer, and if $g(\mu_m(q)) \neq 0$ it can be immediately returned as $f(q)$. We then have to store mapping f restricted only to k-mers from ambiguous buckets, which we denote $\tilde{f}$. Both mappings g and $\tilde{f}$ are stored using BCSFs.

4.2 Cascading

An intermediate layer corresponding to a minimizer length $m < k$, introduced in Section 4.1, can be viewed as a "filter" providing values for some k-mers and "propagating" the other k-mers to the next layer. Therefore, both implementations can be cascaded into more than one layer. This construction is reminiscent of the BBHash algorithm [22] or to cascading Bloom filters from [31].

For $m_1 < m_2 < ...m_\ell \leq k$, each layer i is then input some map f_{i-1} defined on a subset of k-mers $K_{i-1} \subseteq K$ ($f_0 = f$, $K_0 = K$) and outputs another map f_i defined on a smaller subset $K_i \subseteq K_{i-1}$. Each layer stores a bucket table for minimizers $M_{m_i}(K) = \{\mu_{m_i}(q) \,|\, q \in K_{i-1}\}$. The specific definition of f_i and K_i depends on the implementation.

The multi-layer scheme is particularly intuitive for the AMB implementation, where each layer stores a unique value for non-ambiguous minimizers and a special value 0 otherwise. In this case, K_i consists of those k-mers of K_{i-1} hashed to ambiguous buckets, and f_i is simply a restriction of f to those k-mers. Algorithm 3 shows a pseudo-code of multi-level AMB extended to the approximate case (see Section 5 below). The multi-layer version of the FIL scheme is shown in Appendix (Algorithm 4).

Algorithm 3 AMB multi-layer construction algorithm. Exact AMB can be obtained by setting $\delta = 0$.

```
Data: A mapping f of keys to (integer) values, a list L of minimizer lengths, a
      maximum absolute error δ
Result: One BCSFs for each layer
Sort L by increasing order;
T = f;
for m in L do
    Initialise an array A of buckets;
    for (q, c) in T do
        z = μ_m(q);
        if z in A then (r_min, r_max) = g(z) ;
        else (r_min, r_max) = (∞, −∞) ;
        g(z) = (min(r_min, c), max(r_max, c));
    end
    for b = (r_min, r_max) in A do
        if r_max − r_min > δ then b = 0 ;
        else b = r_min ;
    end
    Compress A by using BCSF;
    Create output table O;
    for (q, c) in T do
        if g(μ_m(q)) = 0 then
            Write q and c to O;
        end
    end
    T = O;
end
```

5 Extension to approximate counts

In addition to cascading, the AMB implementation can also be easily extended to work as an approximation algorithm. Consider, to this end, the layered bucketing procedure desribed in 4.2. In the exact case, a bucket is marked as colliding whenever it contains two or more distinct count values. In the approximate case, a collision is defined if a bucket contains a pair of counts, c_i, c_j such that $|c_i - c_j| > \delta$ with δ a pre-defined maximum absolute error. With this modification, the algorithm guarantees to output a value within the absolute error δ from the true count.

Implementing this modification is simple when the majority is computed with the Boyer-Moore majority vote algorithm. Another option is to define $g(s)$ to be the minimum, instead of majority. The rationale of using minimum is the decreasing behavior of k-mer spectra which implies that smaller counts are more frequent and therefore more likely to constitute the majority. It is then sufficient to only remember the maximum $max(s)$ and minimum $min(s)$ values seen by each bucket and check if $max(s) - min(s) > \delta$. If that is the case, then the bucket is marked as colliding, otherwise $min(s)$ is chosen as representative. The latter solution is reported in Algorithm 3.

6 Experimental results

We report experiments on three datasets, two with lower and one with higher empirical entropy. The first is the k-mer counts computed on the *Sakai* strain of *E.Coli* from [36] (NCBI accession number B000007). The second one is a full genome of *C.Elegans*, strain *Bristol N2* downloaded from RefSeq (accession number GCF_000002985.6). The last one is the whole dataset from the same paper [36], hereafter referred to as "df", of k-mer "document frequencies" across 29 *E.Coli* genomes made of approximately 25 million k-mers. Here the document frequency of a k-mer is the number of genomes containing this k-mer.

Experiments were performed on a machine equipped with an Intel® Core™ i7-4770k (Haswell), 8 GB of RAM and Kubuntu 18.04. All construction code is written in python, except for the CSF part which is handled by a simple Java program using Sux4J [11]. Time measurements are performed by a program written in C using the code provided by Sux4J for reading and querying its CSFs. We use xxHash[1] as $\mu_m(q)$ to define an ordering over the minimizers of a given k-mer q. All code is available at `https://github.com/yhhshb/locom.git`.

We only report the best methods for each case with the following naming convention:

- CSF: baseline CSF implementation from Sux4J.
- BCSF: extended CSF with Bloom Filter from Section 3. It may get reduced to a simple CSF if the Bloom Filter is not useful.
- FIL m_1 k: our *first implementation*, saving into each bucket a majority-selected representative and saving corrections into its second layer.
- FIL m_1 m_2 k: same as before but with an additional layer.
- AMB m_1 k: our *second implementation*, selecting each representative by minimum and marking colliding buckets with a special value.
- AMB m_1 m_2 k: same as before but with an additional layer.

In order to apply equation (4), we have to have estimates of C_{BF} and C_{CSF}, that is, estimates of the number of bits per element taken by our implementations of Bloom filter and CSF. For C_{BF}, we have $C_{BF} = 1.44$ corresponding to the theoretical coefficient. For C_{CSF}, we empirically estimated the value as a function of the empirical entropy H_0 of experimental data, and obtained the following estimate:

$$C_{CSF} = \begin{cases} 0.22H_0^2 + 0.18H_0 + 1.16, & \text{if } H_0 < 2 \\ 1.1H_0 + 0.2, & \text{otherwise.} \end{cases} \tag{6}$$

To better understand how different minimizer lengths affect the final compression ratio, we ran FIL and AMB on all possible combinations of 2 and 3 minimizers lengths for $k =$ 10,11,12,13,15,18,21.

6.1 Compression of skewed data

Figure 1 reports memory usage for $k = 13, 15, 18, 21$, when compressing the Sakai dataset. As mentioned earlier, simple CSF takes more than 1 bit/k-mer, which is considerably larger than the entropy of our data. Bloom-enhanced CSF (BCSF) considerably reduces space bringing it closer to the entropy value. For relatively small k's ($k = 13$) AMB and FIL give almost the same results as BCSF, that is, bucketing is not helpful. For larger k's, however,

[1] `https://github.com/Cyan4973/xxHash`

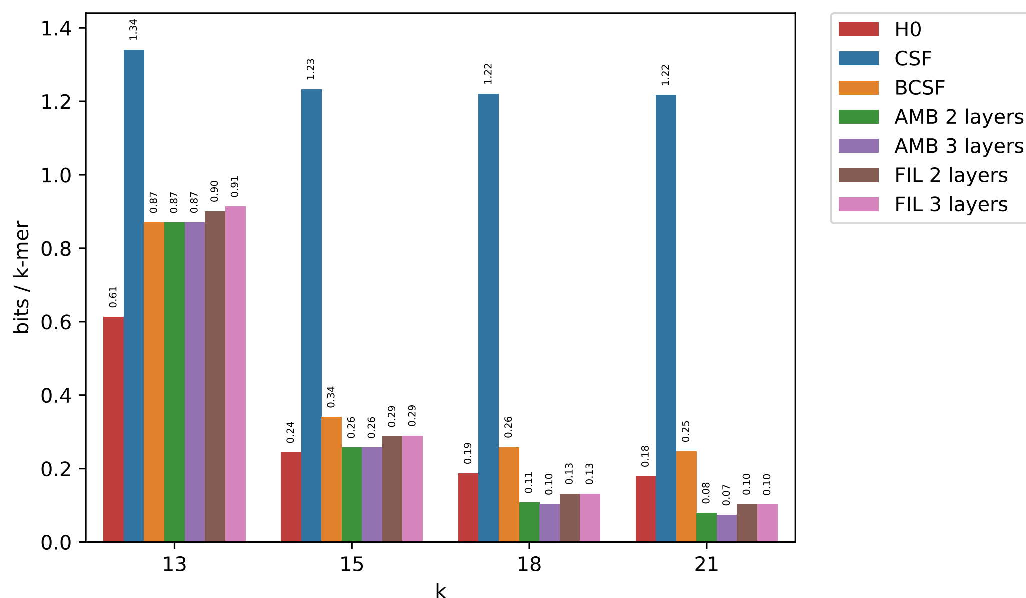

Figure 1 Results for the Sakai dataset for big values of k. For presentation purposes, H0 is represented as an additional red column in each subgroup.

minimizer-based schemes, AMB and FIL, lead to a reduction of space, eventually breaking the entropy barrier for larger values of k ($k = 18, 21$). This demonstrates that for larger k's, minimizers provide an effective way of factoring the space of k-mers in such a way that k-mers with equal counts tend to have the same minimizer.

More in detail, for larger k, the overwhelming majority of buckets are unambiguous (e.g. more than 99% of them, for $k = 18, m = 13$). As a consequence, AMB "filters out" a very large number of k-mers at the first layer, propagating to the second layer only a small fraction of them – those corresponding to ambiguous buckets. The special collision value then becomes the dominant value of the bucket table, making it highly compressible with BCSF. Note also that due to the skewedness of the distribution, i.e. the prevalence of one value, the k-mer tables of the last layer are well compressible as well. Altogether, this enables breaking the empirical entropy lower bound. The situation is similar for FIL: its first layer is even better compressible than the one of AMB, due to the absence of the additional special value which makes the table of AMB slightly less compressible. On the other hand, the BCSF of the second layer table of FIL turns out to take more space than that of AMB. This is because its Bloom filter operates on the large set of all k-mers, which implies a very small value of ε to keep the set of false positives under control, and as a consequence, a relatively large Bloom filter. Overall, FIL turns our to yield a slightly larger space.

The advantages of AMB and FIL tend to vanish for smaller values of k. For small k's, none of the methods beats the empirical entropy, which means that minimizers do not provide an efficient mean to factor the space of k-mers according to count values. We observe that in this case, applying BCSF to the input table provides the most efficient solution.

Since longer k-mers lead to more skewed data, and by extension, to a smaller entropy, both AMB and FIL better compress whole genome count tables for increasing ks. In order to test our algorithms in a more complex and challenging situation, we chose to compress the reference genome of *C.Elegans* (around 100Mbp). Taking into account the considerations

presented previously, we quickly found that the best results for AMB and FIL were given by $k = 18$ when using 2 layers. We randomly chose $m_1 = 19$ and $m = 21$ for three-layer AMB and FIL, respectively. Figure 2 demonstrates that our algorithms are not limited to bacterial genomes. Larger values of k only reduce the entropy of the data, leading to more succinct representations whereas simple CSF could not go below 1.2 bits/k-mer.

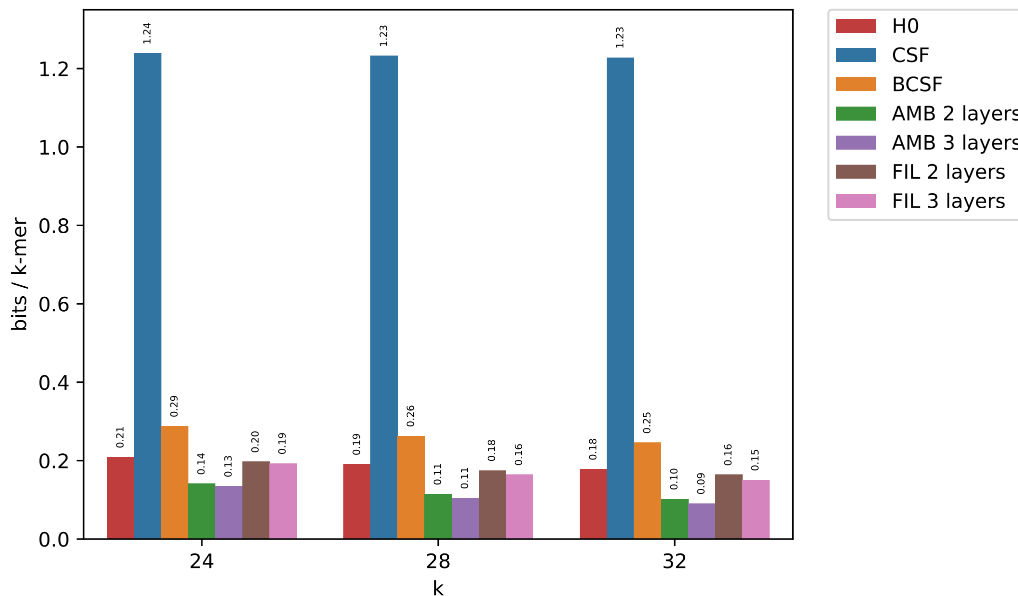

Figure 2 Results when compressing the reference genome of *C.Elegans*.

6.2 Compression of higher entropy data

With very skewed data, collisions of k-mer counts may happen between unrelated k-mers simply because one counter value strongly dominates the spectrum. In order to demonstrate that minimizers are useful as well for less skewed distributions than whole genome count tables, we applied our methods to the df dataset, see Figure 3.

The use of minimizers for larger k's, proves to be beneficial again, with AMB and FIL requiring much less space that the empirical entropy of the data. A similar scenario to the previous case represents itself for relatively small k ($k = 13$), for which both AMB and FIL do not have an advantage over a simpler (B)CSF. For even smaller k-mers (B)CSF remains the best option (results not shown).

The seemingly erroneous exceptions (BCSF taking more space than simple CSF) are explained by the approximation carried by formula (2) (assumption of equal values of C_{CSF} in both sides).

6.3 Approximate counts

In many applications, it is acceptable to tolerate a small absolute error in retrieved counts. Figure 5 shows that, in such case, it is possible to achieve a better memory consumption than simple CSFs even for small values of k. For medium values of k, neither too small nor too big, approximation can lead to the smallest compressed size, even when $\delta = 1$.

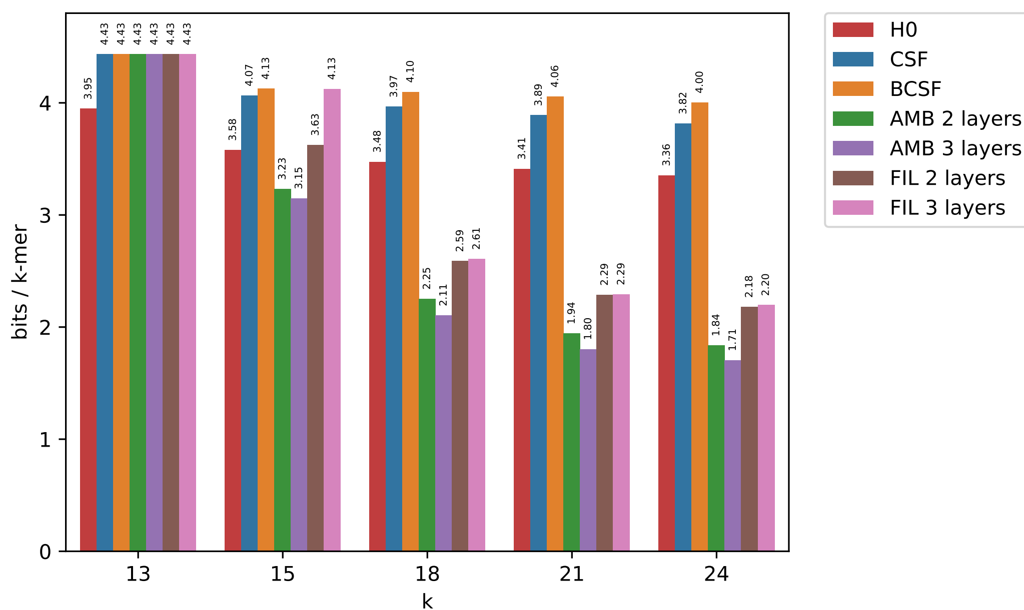

Figure 3 Compressed space usage for the high entropy df dataset.

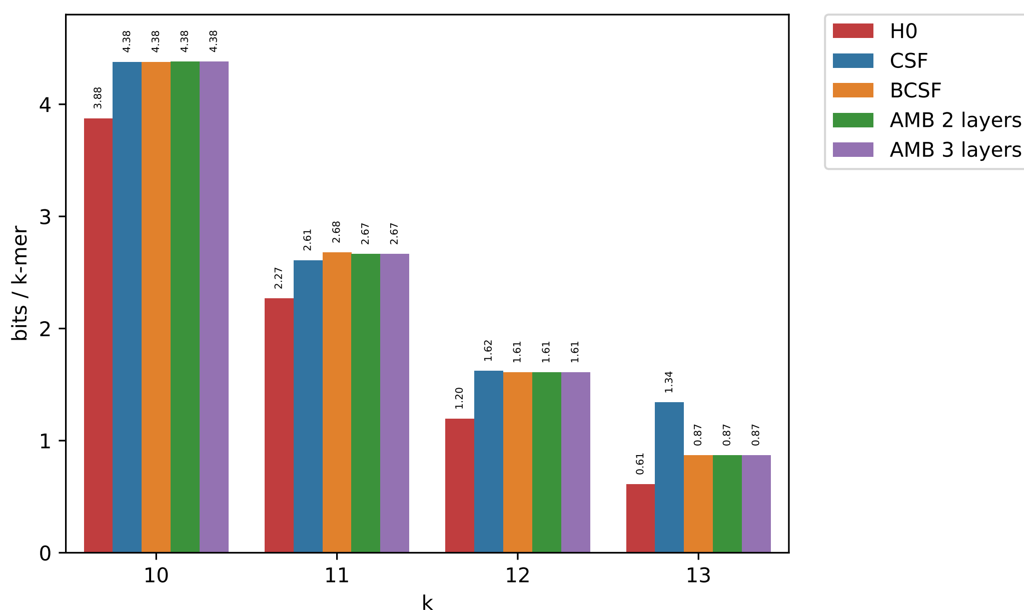

Figure 4 Space usage for the Sakai dataset with small k when using AMB (FIL is slightly worse and was omitted). Minimizer lengths vary between 1 and 5 indicating that the best option is to use a simple (B)CSF.

Unlike Figure 4, reported here for comparison, Figure 5 does not use the best minimizer lengths found for AMB with two layers. This is because we want to use δ to remove ambiguity from as many buckets as possible in each layer, by ignoring small collisions. For small k's this would not be possible with the best solutions found before, because minimizer lengths

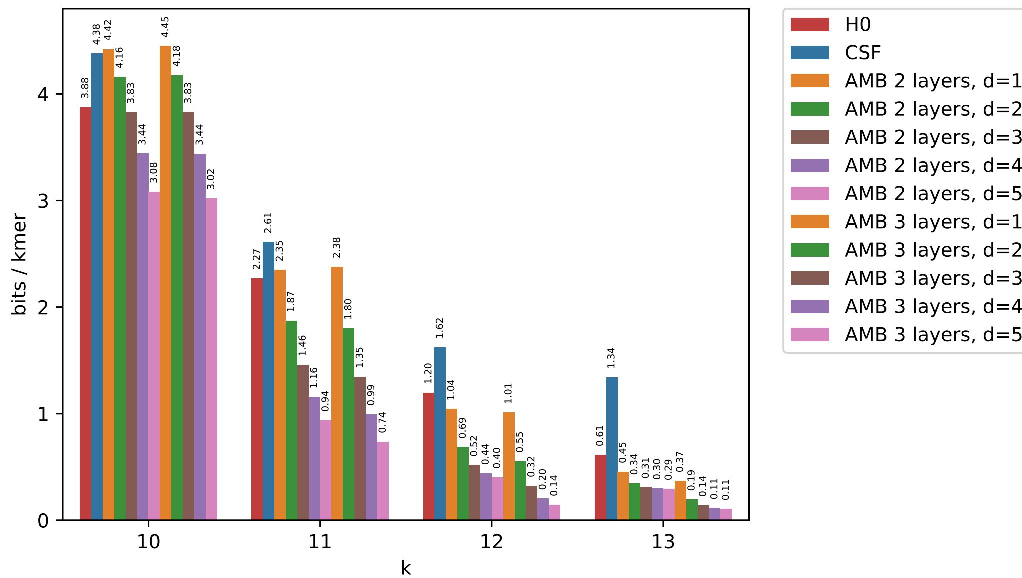

Figure 5 Space usage when using the approximated version of AMB. Entropy (red columns) and CSF (blue columns) are reported for comparison. Unlike Figure 4, AMB is able to break the empirical entropy lower bound when small errors are acceptable.

are too small to allow unambiguity even for $\delta > 0$. Therefore, in this case only, we just use contiguous minimizer lengths for each layer (e.g. if $k = 10$, layers will be $8, 9, 10$ for three-layer AMB).

Another interesting observation about the approximate case is that AMB with three layers is substantially better than AMB with two layers only for $k = 12$ and $k = 13$. For $k = 10$ and $k = 11$ both versions give almost the same results.

6.4 Query speed

Figure 6 shows query time averaged over all distinct k-mers, in ns/k-mer. Simple CSFs, not surprisingly, are the fastest method, with BCSF having a negligible effect on the average query speed. On the other hand, bucketing has a tangible effect on performance, with speed negatively affected by additional layers. For short k-mers, both FIL and AMB are slower than the simple CSF by a factor equal to their number of layers.

The situation is different for larger k's where AMB is only marginally slower than a bare-bones CSF. This is because most queries are solved without accessing all layers every time, thanks to unambiguous buckets. Two layered FIL, on the other hand, gives almost constant average query times across all test, since all queries have to access both of its layers to reconstruct the exact count value. We did not perorm tests for FIL with 3 layers because it will always be slower than the two layered version.

6.5 Choosing minimizer lengths

In all reported cases, good minimizer lengths for the first layer (m_0) follow the rule: $m_0 > m_s = (\log_4(|G|) + 2)$ with $|G|$, the size in base pair of the genome. Smaller m_0, are no longer capable of partitioning k-mers in a meaningful way. Furthermore, space tends to first monotonically decrease to a minimum for increasing minimizer lengths, to increase again

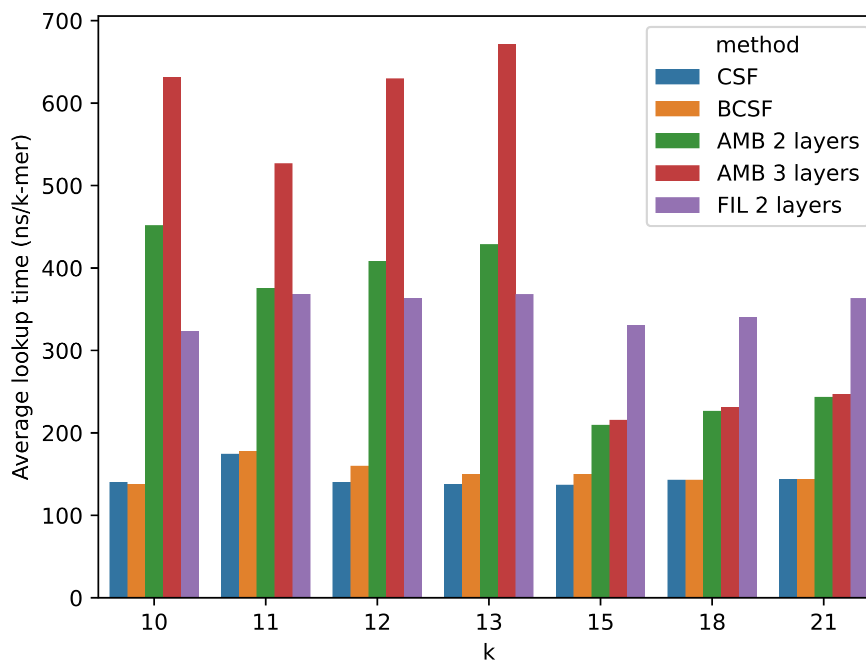

Figure 6 Average query time for AMB with 2 and 3 layers and FIL with 2 layers.

once the optimal value is passed. It is therefore possible to find the minimum by sequentially trying all possible minimizers greater than m_s and stop as soon as the compressed size starts to increase again.

Our results also show how multiple layers have a marginal effect on final compression sizes. In case of AMB, using three layers is always helpful, compared to the two-layer case. Best results are usually achieved for combinations including the best minimizer length obtained for the two-layer case.

On the other hand, FIL with three layers seems to be advantageous only for low entropy data, performing worse that its two-layer counterpart on the df dataset and for small k's.

7 Conclusions

In this work, we introduced three data structures to represent compressed k-mer count tables. Our BCSF algorithm combines Compressed Static Functions, as implemented in Sux4J software [11], with Bloom Filters. This allows for a much better compression for skewed distributions with empirical entropy smaller than 1. Note that to our knowledge, this is the first time that CSFs are used in bioinformatic applications. We also provide a method to dimension the Bloom filter in a BCSF in order to minimise the final space.

Our two other algorithms, FIL and AMB, pair BCSF with a bucketing procedure where count values are mapped into buckets according to minimizer values of respective k-mers. This locality-sensitive hashing scheme allows us to efficiently factor the space of counts, which leads to breaking the empirical entropy lower bound for large enough k's. FIL and AMB use slightly different strategies in decomposing the input table across minimizer layers.

Our last contribution is an extension of AMB to the approximate case, gaining more space at the expense of a small and user-definable absolute error on the retrieved counts.

We validated our algorithms on three different datasets, two fully assembled genomes (*E.Coli* and *C.Elegans*), and one document frequency example, for different k-mer lengths showing how BCSF, AMB and FIL behave in different situations. FIL and AMB have a clear advantage when minimizers are long enough to bucket k-mers in a meaningful way, for both skewed and high entropy data. When it is not possible to define a long-enough minimizer length, the advantage of using intermediate minimizer layers vanishes, and simple CSF and its BCSF provide a better solution.

At query time, CSF and BCSF are the fastest methods requiring about 100ns on average for a single query. For a fixed number of layers, AMB is faster than FIL in all situations when minimizers are useful. FIL becomes faster than AMB only for those cases when both algorithms achieve worse compression ratios than simple (B)CSF.

We consider this study to be the first step towards designing efficient representations for k-mer count tables occurring in data-intensive bioinformatics applications. One possible future direction is compression of RNA-Seq experiments where counts may translate expression levels of genes. Another example is metagenomics where different species may be present with different abundances which can be captured by k-mer counts. In such applications, efficient representation of k-mer counts can be particularly beneficial.

References

1. Djamal Belazzougui and Rossano Venturini. Compressed Static Functions with Applications. In *Proceedings of the 2013 Annual ACM-SIAM Symposium on Discrete Algorithms*, Proceedings, pages 229–240. Society for Industrial and Applied Mathematics, 2013. `doi:10.1137/1.9781611973105.17`.
2. Robert S. Boyer and J. Strother Moore. MJRTY – A Fast Majority Vote Algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–117. Springer Netherlands, Dordrecht, 1991. `doi:10.1007/978-94-011-3488-0_5`.
3. A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, June 1997. `doi:10.1109/SEQUEN.1997.666900`.
4. Benny Chor, David Horn, Nick Goldman, Yaron Levy, and Tim Massingham. Genomic dna k-mer spectra: models and modalities. *Genome Biology*, 10(10):R108, October 2009. `doi:10.1186/gb-2009-10-10-r108`.
5. Yingnan Cong, Yao-ban Chan, and Mark A. Ragan. A novel alignment-free method for detection of lateral genetic transfer based on TF-IDF. *Scientific Reports*, 6(1):30308, 2016. `doi:10.1038/srep30308`.
6. M. Csűrös, L. Noé, and G. Kucherov. Reconsidering the significance of genomic word frequencies. *Trends in Genetics*, 23(11):543–546, November 2007. `doi:10.1016/j.tig.2007.07.008`.
7. Thomas Dencker, Chris-André Leimeister, Michael Gerth, Christoph Bleidorn, Sagi Snir, and Burkhard Morgenstern. Multi-SpaM: A Maximum-Likelihood Approach to Phylogeny Reconstruction Using Multiple Spaced-Word Matches and Quartet Trees. In Mathieu Blanchette and Aïda Ouangraoua, editors, *Comparative Genomics*, Lecture Notes in Computer Science, pages 227–241, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-030-00834-5_13`.
8. Barış Ekim, Bonnie Berger, and Yaron Orenstein. A Randomized Parallel Algorithm for Efficiently Finding Near-Optimal Universal Hitting Sets. In Russell Schwartz, editor, *Research in Computational Molecular Biology*, Lecture Notes in Computer Science, pages 37–53, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-45257-5_3`.
9. Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. RecSplit: Minimal Perfect Hashing via Recursive Splitting. *arXiv:1910.06416 [cs]*, November 2019. `arXiv:1910.06416`.

10 Huan Fan, Anthony R. Ives, Yann Surget-Groba, and Charles H. Cannon. An assembly and alignment-free method of phylogeny reconstruction from next-generation sequencing data. *BMC Genomics*, 16(1):522, July 2015. doi:10.1186/s12864-015-1647-5.

11 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Information and Computation*, 273:104517, August 2020. doi:10.1016/j.ic.2020.104517.

12 Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

13 Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. MetaGraph: Indexing and Analysing Nucleotide Archives at Petabase-scale. *bioRxiv*, page 2020.10.01.322164, November 2020. doi:10.1101/2020.10.01.322164.

14 Mikhail Karasikov, Harun Mustafa, Amir Joudaki, Sara Javadzadeh-no, Gunnar Rätsch, and André Kahles. Sparse Binary Relation Representations for Genome Graph Annotation. *Journal of Computational Biology*, 27(4):626–639, December 2019. doi:10.1089/cmb.2019.0324.

15 Parsoa Khorsand and Fereydoun Hormozdiari. Nebula: Ultra-efficient mapping-free structural variant genotyper. *bioRxiv*, page 566620, March 2019. doi:10.1101/566620.

16 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, May 2017. doi:10.1093/bioinformatics/btx304.

17 Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, 2017. doi:10.1101/gr.215087.116.

18 Téo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. kmtricks: Efficient construction of Bloom filters for large sequencing data collections. *bioRxiv*, page 2021.02.16.429304, 2021. doi:10.1101/2021.02.16.429304.

19 Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, July 2016. doi:10.1093/bioinformatics/btw152.

20 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

21 Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. *Bioinformatics*, 36(5):1374–1381, 2020. doi:10.1093/bioinformatics/btz102.

22 Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SEA.2017.25.

23 Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, July 2020. doi:10.1093/bioinformatics/btaa487.

24 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764, March 2011. doi:10.1093/bioinformatics/btr011.

25 Marmar Moussa and Ion I. Măndoiu. Single cell RNA-seq data clustering using TF-IDF based methods. *BMC Genomics*, 19(6):569, 2018. doi:10.1186/s12864-018-4922-4.

26 Harun Mustafa, André Kahles, Mikhail Karasikov, and Gunnar Rätsch. Metannot: A succinct data structure for compression of colors in dynamic de Bruijn graphs. *bioRxiv*, page 236711, March 2018. doi:10.1101/236711.

27 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and Perfect Hashing Using Fingerprinting. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, Lecture Notes in Computer Science, pages 138–149, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07959-2_12.

28 Atif Rahman, Ingileif Hallgrímsdóttir, Michael Eisen, and Lior Pachter. Association mapping from sequencing reads using k-mers. *eLife*, 7:e32920, June 2018. `doi:10.7554/eLife.32920`.

29 Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, March 2013. `doi:10.1093/bioinformatics/btt020`.

30 Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. `doi:10.1093/bioinformatics/bth408`.

31 K. Salikhov, G. Sacomoto, and G. Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *BMC Algorithms for Molecular Biology*, 9(1):2, 2014. URL: `http://www.almob.org/content/9/1/2`.

32 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, San Diego, California, 2003. Association for Computing Machinery. `doi:10.1145/872757.872770`.

33 Moustafa Shokrof, C. Titus Brown, and Tamer A. Mansour. MQF and buffered MQF: Quotient filters for efficient storage of k-mers with their counts and metadata. *bioRxiv*, page 2020.08.23.263061, August 2020. `doi:10.1101/2020.08.23.263061`.

34 Gregory E. Sims, Se-Ran Jun, Guohong A. Wu, and Sung-Hou Kim. Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. *Proceedings of the National Academy of Sciences of the United States of America*, 106(8):2677–2682, February 2009. `doi:10.1073/pnas.0813249106`.

35 Derrick E. Wood and Steven L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, March 2014. `doi:10.1186/gb-2014-15-3-r46`.

36 Huiguang Yi and Li Jin. Co-phylog: an assembly-free phylogenomic approach for closely related organisms. *Nucleic Acids Research*, 41(7):e75, 2013. `doi:10.1093/nar/gkt003`.

37 Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *arXiv:1608.05699 [cs]*, November 2017. `arXiv:1608.05699`.

38 Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. SeqOthello: querying RNA-seq experiments at scale. *Genome Biology*, 19(1):167, 2018. `doi:10.1186/s13059-018-1535-9`.

39 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Lower Density Selection Schemes via Small Universal Hitting Sets with Short Remaining Path Length. In Russell Schwartz, editor, *Research in Computational Molecular Biology*, Lecture Notes in Computer Science, pages 202–217, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-45257-5_13`.

A Multilayer FIL algorithm

Algorithm 4 FIL multi-layer construction algorithm.

```
Data: A mapping f of keys to (integer) values, a list L of minimizer lengths
Result: One BCSFs + Bloom filter for each layer
Sort L by increasing order;
T = f;
for m in L do
    Initialise an array A of buckets;
    n = 0;
    for (q, c) in T do
        z = μ_m(q) Insert c into g(z);
        n = n + 1;
    end
    for b in A do
        Select representative r of bucket b by majority rule;
    end
    Compress A by using BCSF;
    Create output table O;
    p_q = 0;
    for (q, c) in T do
        if g(μ_m(q)) ≠ c then
            Write q and c − g(μ_m(q)) to O;
            p_q = p_q + 1;
        end
    end
    α = (n − p_q)/n;
    ϵ = (1 − α)/α;
    if ϵ < 1 then
        Initialise an empty Bloom Filter of size 1.44 log_2(1/ϵ);
        Insert all elements of O into B;
        for (q, c) in T do
            if g(μ_m(q)) = c and B(q) then
                Write q and c − g(μ_m(q)) to O
            end
        end
    end
    T = O;
end
```

B Additional figures

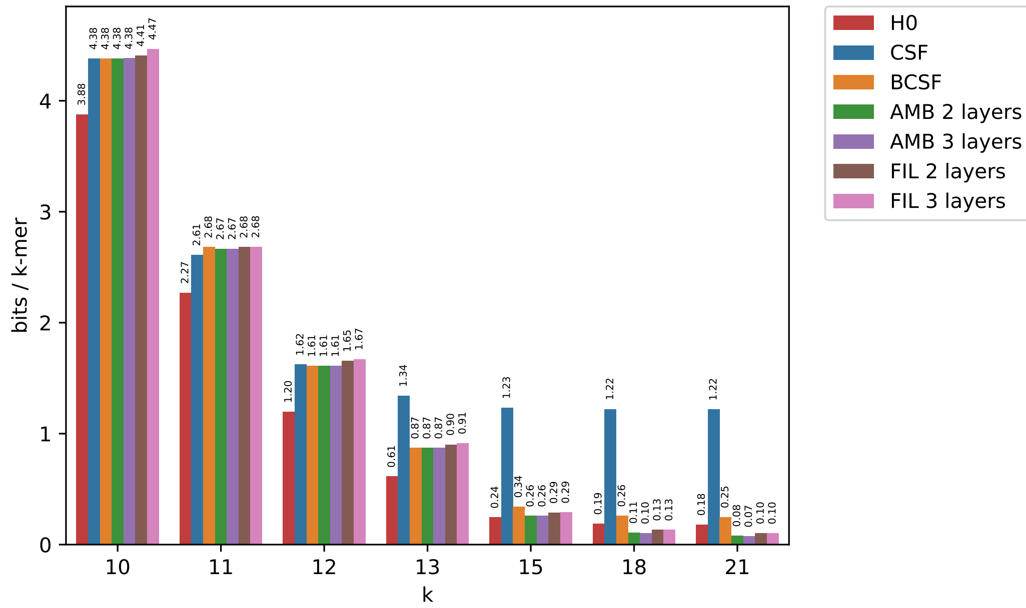

Figure 7 Space usage across all values of k, for the Sakai dataset.

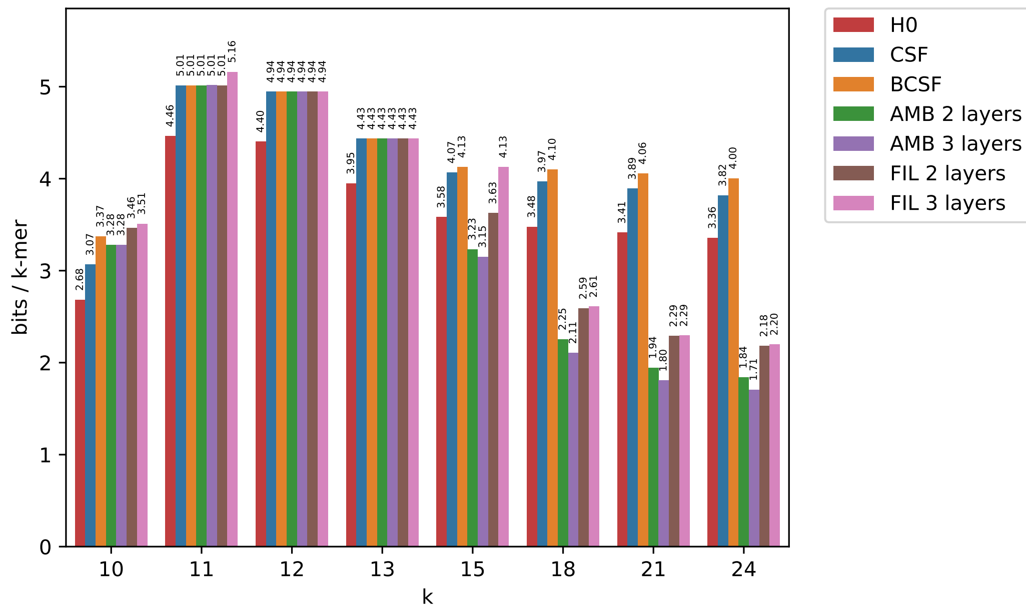

Figure 8 Space usage across all values of k, for the df dataset.