DATA STRUCTURES TO REPRESENT SETS OF k-LONG DNA SEQUENCES

RAYAN CHIKHI, JAN HOLUB, AND PAUL MEDVEDEV

Abstract. The analysis of biological sequencing data has been one of the biggest applications of string algorithms. The approaches used in many such applications are based on the analysis of k-mers, which are short fixed-length strings present in a dataset. While these approaches are rather diverse, storing and querying k-mer sets has emerged as a shared underlying component. Sets of k-mers have unique features and applications that, over the last ten years, have resulted in many specialized approaches for their representation. In this survey, we give a unified presentation and comparison of the data structures that have been proposed to store and query k-mer sets. We hope this survey will not only serve as a resource for researchers in the field but also make the area more accessible to outsiders.

1. Introduction

String algorithms have found some of their biggest applications in modern analysis of sequencing data. Sequencing is a type of technology that takes a biological sample of DNA or RNA and extracts many *reads* from it. Each read is a short substring (e.g. 250 characteres) of the original sample, subject to errors. Analysis of sequencing data relies on string matching with these reads, and many popular methods are based on first identifying short, substrings of the reads (called k-mers, where k refers to the length of the substring). Such k-mer-based methods have become more popular in the last ten years due to their inherent scalability and simplicity. They have been applied across a wide spectrum of biological domains, e.g. genome and transcriptome assembly, transcript expression quantification, metagenomic classification, structural variation detection, and genotyping. While the algorithms working with k-mers are rather diverse, storing and querying sets of k-mers has emerged as a shared underlying component. Because of the massive size of these sets, minimizing their storage requirements and query times is becoming its own area of research.

Representing a set is a well-studied problem in computer science. However, the fact that the set consists of strings lends structure that can be exploited for efficiency. Even beyond this, there are aspects of k-mer sets that have led to the development of specialized approaches. Let S denote a set of n k-mers. First, in most applications, the alphabet has constant size, denoted by σ. Second, most applications revolve around *sparse* sets (i.e. $n = o(\sigma^k)$). Third, n is typically much larger than k, e.g. k is usually between 20 and 200, while n can be in the billions.

Another unique aspect of k-mer sets is what we call the *spectrum-like-property*. S has the *spectrum-like-property* if there exists an underlying but unknown collection $\mathcal{G}$ of long strings such that S contains a significant portion of the k-mers of $\mathcal{G}$, and, conversely, many of the k-mers of S are either exact or "noisy" substrings of $\mathcal{G}$. For example, sequencing a metagenome sample ($\mathcal{G}$ would be the set of abundant genomes in this case) generates a set of reads, which cover most of the abundant genomes in the sample. A computational tool would then chop the reads up into their constituent k-mers (e.g. $k = 50$), and store these in the set S. Some other examples of $\mathcal{G}$ are a single genome (e.g. whole genome sequencing), a collection of transcripts (RNA-seq or Iso-Seq), or enriched genomic regions (e.g. ChIP-seq). We introduce this property in order to informally

This research has started during J. Holub's research stay at the Pennsylvania State University supported by the Fulbright Visiting Scholar Program and it was finished with the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics". This work was partially supported by NSF awards DBI-1356529, CCF-551439057, IIS-1453527, and IIS-1421908 to PM. Research reported in this publication was supported by the National Institute Of General Medical Sciences of the National Institutes of Health under Award Number R01GM130691, and the INCEPTION project (PIA/ANR-16-CONV-0005). The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

capture an important aspect of S in many applications that arise from sequencing. This property is exploited by methods for representing k-mer sets and also drives the types of queries that are performed on them. Our definition is necessarily imprecise, in order to capture the huge diversity within applications.

In this survey, we give a unified presentation and comparison of the data structures that have been proposed to store and query k-mer sets. We focus on key ideas and contributions and refer the reader to the original papers for technical details. We hope this survey will not only serve as a resource for researchers in the field but also make the area more accessible to outsiders.

2. Operations

In this section, we describe the type of operations supported by data structures representing S. We will assume that the size of the alphabet (σ) is constant, all logs are base 2, strings are 1-indexed, and S is sparse. The most basic operations that a data structure representing S should support are its construction and checking whether a k-mer x is in S (*memb*). If the data structure is *dynamic*, it also supports inserting a k-mer into S (*insert*) or deleting a k-mer from S (*delete*). A data structure where insertion and deletion is either not possible or would require as much time as re-construction is called *static*.

Recall that in the context of spectrum-like-property, there is an underlying set of strings $\mathcal{G}$ that is generating the k-mers of S. This implies that many k-mers in S will have *dovetail* overlaps with each other (i.e. the suffix of one k-mer equals to the prefix of another), often by $k-1$ characters. Algorithms that use S in order to reconstruct $\mathcal{G}$ often work by starting from a k-mer and extending it one character at a time to obtain the strings of $\mathcal{G}$. This motivates having efficient support for operations that check if an extension of a k-mer exists in S. Formally, given $x \in S$ and a character a, the $fwd(x, a)$ operation returns true if $x[2, k] \cdot a$ is in S (we use the notation $x[i, j]$ to refer to the substring of x starting from the i^{th} character up to and including the j^{th} character). Similarly, the $bwd(x, a)$ operation checks whether $a \cdot x[1, k-1]$ is in S.

We assume that a data structure maintains some kind of internal state corresponding to the last queried k-mer i.e. a $memb(x)$ query would leave the data structure in a state corresponding to x, a $fwd(x, a)$ query would leave the state corresponding to $x[2, k] \cdot a$, etc. For example, for a hash table, the internal state after a $memb(x)$ query would correspond to the hash value of x and to the location of x in the table; in the case of a BWT-type data structure, the internal state corresponds to an interval representing x.

We also assume that prior to a call to $fwd(x, a)$ or $bwd(x, a)$, the data structure is in a state corresponding to x. In this way, $fwd(x, a)$ and $bwd(x, a)$ are different from $memb(x[2, k] \cdot a)$ and $memb(a \cdot x[1, k-1])$, respectively. However, for data structures that do support *fwd* or *bwd* explicitly or do not maintain an internal state, there is always the default implementation using the corresponding membership query.

3. Basic approaches

Perhaps the most basic static representation that is used in practice is a lexicographically **sorted list** of k-mers. The construction time is $O(nk)$ using a linear time string sort algorithm and the space needed to store the list is $\Theta(nk)$. A membership query is executed as a binary search in time $O(k \log n)$. This representation is both space- and time-inefficient but can be used by someone with very limited computer science background, making it still relevant.

There are traditional data structures to represent sets of elements that do not take advantage of the fact that the elements are strings. One such data structure is a binary search tree and its variants. However, a binary search tree requires $O(\log n)$ time for membership queries and is in most aspects worst than a string trie (Mäkinen et al., 2015). It is rarely used in the context of DNA sequences. Another such data structure is the **hash table**, where a membership query, insertion, deletion, *fwd* and *bwd* require $O(k)$ amortized time, which is the time needed to hash a k-mer. Rolling hash functions can improve this to $\mathcal{O}(1)$ for *fwd*/*bwd* queries or for the case when the *memb* queried k-mers are consecutive substrings of a bigger string (e.g. a read). These fast times and the wide availability of high-quality hash table libraries make them popular in

some applications. However, a hash table requires $\Theta(nk)$ space, which is prohibitive for large applications due to the k factor.

Conway and Bromage (Conway and Bromage, 2011) were one of the first to consider more space-efficient representations of k-mer sets. S can be thought of as a binary bitvector of length σ^k, where each k-mer corresponds to a position in the bitvector and the value of the bit reflects whether the k-mer is present in S. Since S is sparse, storing the bitvector wastes a lot of space, but a sparse bitmap representation (Okanohara and Sadakane, 2007) based on Elias-Fano coding (Elias, 1974) can be used to store the bitvector; then, the *memb* operation becomes a pair of rank operations (i.e. finding the number ones in a prefix of a bitvector) on the compressed bitvector. However, if S is *exponentially sparse* (i.e. $\exists \epsilon > 0$ such that $n = \mathcal{O}(\sigma^{k(1-\epsilon)})$), the space needed is $\Omega(nk)$.

An approximate membership query data structure is a representation of a set that is space-efficient in exchange for allowing membership queries to occasionally return false positives (no false negatives are allowed though). A false positive occurs when $x \notin S$ but $memb(x)$ returns true. These data structures are applicable whenever space savings outweigh the drawback of allowing some false positives or when the effect of false positives can be mitigated using other methods.

Bloom filters (Bloom, 1970) (abbreviated BF) are a classical example that has found widespread use in representing k-mer sets (see Broder and Mitzenmacher (2004) for a survey). Pell et al. (2012) applied Bloom filters to k-mer sets, supporting *insert*, *memb*, *fwd*, and *bwd* operations in the time it takes to hash a k-mer (usually $\Theta(k)$, except for rolling hash functions). A BF does not support $delete(x)$, though there are variants of BFs that make trade-offs to support it in $\Theta(k)$ time (e.g. counting BFs (Fan et al., 2000), spectral BFs (Cohen and Matias, 2003), and quotient filters (Bender et al., 2012)). BFs can be compressed using RRR encoding (Raman et al., 2007), resulting in a further time-space tradeoff (Mitzenmacher, 2002). Pellow et al. (2016) have shown how to exploit the spectrum-like-property to reduce the false positive rate of BFs.

Bloom filters are popular because they reduce the space usage to $\mathcal{O}(n)$ while maintaining $\mathcal{O}(k)$ membership query time. However, operations on Bloom filters require access to distant parts of the data structure, and therefore do not scale well when they do not fit into RAM. Bloom filters and their variants are valuable for their simplicity and flexibility but more advanced approximate membership data structures offer better performance. In particular, the quotient filter (Bender et al., 2012) and the counting quotient filter (Pandey et al., 2017c) have been applied to storing k-mer sets in the Squeakr tool (Pandey et al., 2017d).

Most of the above data structures do not take advantage of the fact that the elements of S are strings. However, there is a rich literature of string-based indices (Mäkinen et al., 2015), some of which can be modified to store and query k-mer sets. For example, the **FM-index**[1] can be defined and constructed for a set of strings, using the Extended Burrows-Wheeler Transform (Mantaci et al., 2005). A scalable version has been implemented in the BEETL software (Bauer et al., 2013). This can in principle be applied to S, though we are unaware of such an application in practice. In theory, it would result in $\mathcal{O}(nk)$ construction time and $\mathcal{O}(k)$ *memb* query time (Bauer et al., 2013). A naive implementation of *fwd* and *bwd* operations in this setting would require a new *memb* query, though a more sophisticated approach, using bidirectional indices, may improve the run-time.

Another example is the trie data structure and its variations. In the setting of representing a k-mer set, this idea is implemented in the Bloom filter trie (Holley et al., 2016). It combines the elements of Bloom filters and burst tries (Heinz et al., 2002). Conceptually, a small parameter $\ell < k$ is chosen and all the k-mers are split into k/ℓ equal-length parts. The i-th part is then stored within a node at the i-th level of the trie. Bloom filters are used within nodes to quickly filter out true negatives when querying the membership of a k-mer part. The Bloom filter trie offers fast *memb* time ($\mathcal{O}(k)$) but requires $\mathcal{O}(nk)$ space.

[1]We note that the FM-index and its variants are also sometimes referred to as a BWT-indices, since they are based on the Burrows-Wheeler Transform (BWT).

4. De Bruijn graphs

De Bruijn graphs provide a useful way to think about k-mer sets that have the spectrum-like-property and for which *fwd* and *bwd* operations should be supported in $\mathcal{O}(1)$ time. A de Bruijn graph (dBG) is directed graph built from a set of k-mers S. In the *node-centric* de Bruijn graph, the node set is given by S and there is an edge from u to v iff the last $k-1$ characters of u are equal to the first $k-1$ characters of v. In a *edge-centric* de Bruijn graph, the node set is given by the set of $(k-1)$-mers present in S, and, for every $x \in S$, there is an edge from $x[1,k-1]$ to $x[2,k]$. The node-centric dBG of S is a line graph (Bang-Jensen and Gutin, 2008) of the edge-centric dBG of S, and without loss-of-generality, we mostly focus our discussion on node-centric dBGs. We note that the concept of a de Bruijn graph in bioinformatics is originally borrowed from combinatorics, where it is used to denote the node-centric dBG (in the sense we define here) of the full k-mer set, i.e. a set of all σ^k k-mers.

The dBG is a mathematical object constructed from S that explicitly captures the overlaps between the k-mers of S. Since this information is already implicitly present in S, the dBG contains the same underlying information as S. However, the graph formalism gives us a way to apply graph-theoretic concepts, such as walks or connected components, to k-mer sets. In theory, all these concepts could be stated in terms of S directly without the use of the dBG. For example a simple path in the node-centric dBG could be defined as an ordered subset of S such that every consecutive pair of k-mers x and y obey $x[2,k] = y[1,k-1]$. However, using the graph formalism directly makes the use of graph-theoretic concepts simpler and more immediate.

Just like S is a mathematical object that can be represented by various data structures, so is the dBG. In this sense, the term dBG can have a fuzzy meaning when it is used to refer to not just the mathematical object but to the data structure representing it. Generally, though, when a data structure is said to represent the dBG (as opposed to S), it is meant that edge queries can be answered efficiently. When projected onto the operations we consider in this paper, in- and out-edge queries are equivalent to *bwd* and *fwd* queries, respectively. For example, a query to check if x has an outgoing edge to y is equivalent to the $fwd(x, y[k])$ operation, while $fwd(x, a)$ is equivalent to checking if x has an outgoing edge to $x[2,k]a$.

4.1. Node- or edge-based representations.

The simplest data structures that represent graphs are the incidence matrix and the adjacency list. The incidence matrix representation requires $\Theta(n^2)$ space and is rarely used for dBGs (the inefficiency can also be explained by the fact the incidence matrix is not intended for sparse graphs, but the dBG is sparse because its nodes have constant in- and out-degrees of at most σ). A **hash table adjacency list** representation is possible using a hash table that stores, for each node, 2σ bits to signify which incident edges exist in the graph. The extension operations still require the time needed to hash a k-mer because the hash value for the extension needs to be calculated in order to change the "internal state" of the hash table to the extension. However, checking which extensions exist can be done in constant time. While this representation requires $\Theta(nk)$ space, its ease of implementation makes it a popular choice for smaller n or k.

The special structure of dBGs has been exploited to create a more space-efficient representation called **BOSS** (the name comes from the initials of the inventors (Bowe et al., 2012)). BOSS represents the edge-centric dBG as a list of edge labels, sorted by concatenation of node and edge labels. BOSS builds upon the XBW-transform (Ferragina et al., 2009) representation of trees, which itself is an extension of the FM-index (Ferragina and Manzini, 2000) for strings. BOSS further modified the XBW-transform to work for dBGs. Historically, BOSS was initially introduced such that it was computed on a single string as input (Bowe et al., 2012); then an efficient implementation used k-mer-counted input (COSMO, Boucher et al. (2015)); finally some modifications have been made to the original structure for usage in a real genome assembler (Li et al., 2016). BOSS occupies $4n + o(n)$ bits of space and allows operation $memb(u)$ in $\mathcal{O}(k)$ time, which works like the search operation in an FM-index (Ferragina and Manzini, 2000). This assumes that there is only one source and one sink in the dBG. If there are more sources and sinks in the dBG but their number is negligible, the space becomes $5n + o(n)$ (since a separator character as needed, as described in Bowe et al. (2012). Otherwise, in the worst case, the space

needed becomes $\Theta(nk)$ (Bowe et al., 2012; Boucher et al., 2015). In the version given by Li et al. (2016), the space is always $6n + O(1)$, but then membership queries are not always exact. BOSS achieves a $\mathcal{O}(1)$ run-time for the *fwd* operation, while *bwd* still runs in $\mathcal{O}(k)$ time. The *bwd* run time can further be reduced to $\mathcal{O}(1)$ using the method of Belazzougui et al. (2016b), at the cost of $\mathcal{O}(n)$ extra space. This representation is static, but a dynamic one is also possible by sacrificing some query time (Bowe et al., 2012; Belazzougui et al., 2016b).

4.2. **Unitig-based representations.** A *unitig* in a node-centric dBG is a path over the nodes $(x_1, \ldots, x_\ell)$, with $\ell \geq 1$ such that either (1) $\ell = 1$, or (2) for all $1 < i < \ell$, the out- and in-degree of x_i is 1 and the in-degree of x_ℓ is 1 and the out-degree of x_1 is 1. A unitig is *maximal* if the underlying path cannot be extended by a node while maintaining the property of being a unitig. The set of maximal unitigs in a graph is unique and forms a node decomposition of the graph (Lemma 2 in Chikhi et al. (2016)). In the literature, maximal unitigs are sometimes referred to as unipaths or as simply unitigs. Computing the maximal unitigs can also be viewed as a task of compacting together their constituent nodes in the graph; hence this is sometimes referred to as graph compaction.

A maximal unitig $(x_1, \ldots, x_\ell)$ spells a string $s = x_1 x_2[k] \cdots x_\ell[k]$ with the property that a k-mer x is a substring of s iff $x \in \{x_1, \ldots, x_\ell\}$. Thus, the list of maximal unitigs is an alternate representation of the k-mers in S in the sense that $x \in S$ if and only if x is a substring of a maximal unitig of the dBG of S. This representation reduces the amount of space since a maximal unitig represents a set of ℓ k-mers using $k-1+\ell$ characters, while the raw set of k-mers uses $k\ell$ characters. The number of characters taken by the list is $n + U(k-1)$, where U is the number of maximal unitigs. In many bioinformatic applications, U is much smaller than n and this representation can greatly reduce the space. However, since one can always construct a set S with $U = n$, this representation does not yield an improvement when using worst-case analysis.

Given these space savings, one can pre-compute the maximal unitigs of S as an initial, lossless, compression step. This is itself a task that builds upon other k-mer set representations. However, there are fast and low-memory stand-alone tools for compaction such as BCALM (Chikhi et al., 2016) or others (Pan et al., 2018; Guo et al., 2018); more generally, algorithms for compaction are often presented as part of genome assembly algorithms, which are too numerous to cite here.

In order to support efficient *memb*, *fwd*, and *bwd* queries, the maximal unitigs must be appropriately indexed. The **DBGFM** data structure (Chikhi et al., 2014) builds an FM-index of the maximal unitigs in order to allow membership queries. In **deGSM** (Guo et al., 2018), the authors similarly build a BWT of the maximal unitigs; but, they demonstrate how this can be done more efficiently by not explicitly constructing the strings of maximal unitigs. These representations allow for $\mathcal{O}(k)$ membership queries. For a k-mer that is not the first or last k-mer of a maximal unitig, there is exactly one *fwd* and *bwd* extension, and it is determined by the next character in the unitig. For such k-mers, these operations can be done in very small constant time, without the need to use the FM-index. In the case that a k-mer lies at the end of its maximal unitig, it may have multiple extensions and they would be at an extremity of another maximal unitig. In this case a new *memb* query is required, though more sophisticated techniques may be possible to reduce the query times. It should be noted that these approaches, as implemented, are static; however, it may be possible to modify them to allow for insertion and deletion.

Given a static set S of size n, a hash function is perfect if its image by S has cardinality n, i.e. there are no collisions. Furthermore, the hash function is minimal if the image consists of integers smaller or equal to $n-1$. For a k-mer set S, one can construct an minimal perfect hash function (MPHF) in $\mathcal{O}(nk)$ time and store it in cn bits of space where c is a small constant (around 3) (Belazzougui et al., 2009; Limasset et al., 2017); calculating the hash value of a k-mer is done in $\mathcal{O}(k)$ time. There exists an efficient implementation of MPHF for k-mer sets, BBHash (Limasset et al., 2017). The advantage of a MPHF is that one can use it to associate information with each k-mer in S; this is done by creating an array of size n and using the hash value of a k-mer as its index into the array. Unlike a hash table, this requires $\mathcal{O}(n)$ instead of $\mathcal{O}(nk)$ space. The disadvantage of a MPHF is that if it is given a k-mer $x \notin S$, it will still return a location associated with some arbitrary $x' \in S$. Thus it cannot be used to test for

membership without further additions. Furthermore, support for insertions and deletions would require a dynamic perfect hashing scheme, yet to the best of our knowledge the only efficient implementation for large key sets (Limasset et al., 2017) is static. This limitation is inherited by the MPHF-based schemes we will describe in this paper.

The **pufferfish** index (Almodaresi et al., 2018) uses a MPHF as an alternate to the FM-index when indexing the maximal unitigs. The MPHF along with additional information enables mapping each k-mer to its location in the maximal unitigs. To check for membership, a k-mer x is first mapped to its location; then, $x \in S$ if and only if the k-mer at the location is equal to x The pufferfish index is static, because of its reliance on the MPHF.

5. Navigational data structures

Many genome assembly algorithms start from a k-mer in the dBG and proceed to navigate the graph by following the out- and in-neighbor edges. Membership queries are only needed to seed the start of a navigation with a k-mer. Afterwards, only *fwd* and *bwd* queries are performed. In this way, we can continue navigating to all the k-mers reachable from the seed. A data structure to represent S can take advantage of this behavior pattern. Formally, a *navigational data structure* is one where membership queries are either very expensive or impossible, but *fwd* and *bwd* queries are cheap (e.g. $\mathcal{O}(k)$). Navigational data structures were first used by Chikhi and Rizk (2012) and later formalized in Chikhi et al. (2014).

An MPHF in combination with a hash table adjacency list representation of a dBG forms a natural basis for a navigational data structure. This scheme was first described in the literature by Belazzougui et al. (2016a) but was previously implemented in the SPAdes assembler (Bankevich et al., 2012). An MPHF is first built on S and then used to index a direct access table (i.e. an array). Each entry is composed of 2σ bits indicating which incident edges exist. For $x \in S$, we can answer $fwd(x, a)$ and $bwd(x, a)$ queries using the table. It takes only $\mathcal{O}(1)$ time to find out if an extension exists, but the queries take $\mathcal{O}(k)$ time because a hash value has to be computed to actually navigate to the extension. If a rolling MPHF is used, this can also take $\mathcal{O}(1)$ time.

The list of maximal unitigs also forms a natural basis for a navigational data structure, without the need of constructing any additional index to support *memb* queries. As previously described, when maximal unitigs are stored, the *fwd* and *bwd* queries are trivial for most k-mers. The exceptions occur when *fwd* is executed on the last k-mer in a maximal unitig or when *bwd* is executed on the first k-mer in a maximal unitig. These extensions must be stored in a structure separate from the maximal unitigs; for example, the hash table adjacency list indexed by a MPHF can be used as described above. When the number of maximal unitigs is significantly smaller than n, the cost of this additional structure is negligible.

Another approach to constructing a navigational data structure builds on the Bloom filter (BF). A BF is first built to store the k-mers of S, but a hash table is also used to store the k-mers that are false positives in the BF and are extensions of elements of S (Chikhi and Rizk, 2012). This allows to avoid false positives for *fwd*/*bwd* queries by double checking the hash table. More memory efficient approaches use a cascading Bloom filter (Salikhov et al., 2013; Jackman et al., 2016), which is a sequence $B_1, \ldots, B_n$ of increasingly smaller Bloom filters, where B_1 is an initial Bloom filter that stores S and B_i ($i > 1$) records a subset of the false positives of B_{i-1}. BF-based navigational data structures support exact *fwd*/*bwd* queries in $\mathcal{O}(k)$ time (or $\mathcal{O}(1)$ with a rolling hash); as a bonus, they can also support approximate *memb* queries (they do not support *insert* operations). In this sense, they can be viewed as a compromise between navigational and normal data structures that trades exact membership of non-extension k-mers for better space-efficiency. Alternatively, they can be viewed as an augmentation of the simple Bloom filter representation to guarantee that at least the extension queries are exact.

Belazzougui et al. (2016a) proposed a mechanism to transform their navigational data structure into a membership data structure. They give both a static and dynamic version; we present the static one here. They store the MPHF-indexed adjacency list together with a forest of node-disjoint rooted trees that is a node-covering subgraph of the dBG. Each tree has bounded height

data structure	*memb*	*fwd*	*bwd*
sorted list	$k \log n$	[a] $k \log n$	[a] $k \log n$
hash table adj. list	k	[b] 1 or k	[b] 1 or k
Conway and Bromage	$\max(\log \frac{\sigma^k}{n}, \frac{\log^4 n}{k \log \sigma})$	[a] $\max(\log \frac{\sigma^k}{n}, \frac{\log^4 n}{k \log \sigma})$	[a] $\max(\log \frac{\sigma^k}{n}, \frac{\log^4 n}{k \log \sigma})$
Bloom filter[1]	k	[b] 1 or k	[b] 1 or k
Bloom filter trie	k	[a] k	[a] k
BOSS (static)	k	1	1
BOSS (dynamic)	$k(1 + \frac{\log n}{\log \log n})$	$\frac{\log n}{\log \log n}$	$k(1 + \frac{\log n}{\log \log n})$
unitigs-based[2]	k	[c] 1 or k	[c] 1 or k
Belazzougui et al[3]	k	[a] 1	[a] 1

TABLE 1. Query Complexities. Big O notation is implied for all the complexities, but the $\mathcal{O}$ symbol is omitted from the table for clarity.
[a]there is no specialized extension query so the time is the same as for *memb*.
[b]$\mathcal{O}(1)$ occurs if a rolling hash function is used, otherwise there is no specialized extension query.
[c]$\mathcal{O}(1)$ occurs if the extension lies on the same unitig, or, in the case of pufferfish, if a rolling MPHF is used.
[1]the Bloom filter is non-exact and may return false positives.
[2]This includes DBGFM (Chikhi et al., 2014), deGSM (Guo et al., 2018), and pufferfish (Almodaresi et al., 2018).
[3]This includes both the static and dynamic version presented in Belazzougui et al. (2016a). But, the dynamic version may, with low probability, give incorrect query answers.

	construction		modification	
data structure	time	space	*insert*	*delete*
sorted list	$\mathcal{O}(nk)$	$\Theta(nk)$	-	-
hash table adj. list	$\mathcal{O}(nk)$	$\Theta(nk)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Conway and Bromage	$\Omega(nk)$	$\Theta(n(1 + \log \frac{\sigma^k}{n}))$	-	-
Bloom filter	$\mathcal{O}(nk)$	$\mathcal{O}(n)$	$\mathcal{O}(k)$	-
Bloom filter trie	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(k)$	-
BOSS (static)	$\mathcal{O}(nk \frac{\log n}{\log \log n})$	[a] $\mathcal{O}(n)$	-	-
BOSS (dynamic)	$\mathcal{O}(nk \frac{\log n}{\log \log n})$	[a] $\mathcal{O}(n)$	$\mathcal{O}(k \frac{\log n}{\log \log n})$	-
unitigs-based	$\mathcal{O}(nk)$	$\mathcal{O}(n + U(k-1))$	-	-
Belazzougui et al (static)	$\mathcal{O}(nk)$	[b] $\mathcal{O}(n + kC)$	-	-
Belazzougui et al (dynamic)	$\mathcal{O}(nk)$	[b] $\mathcal{O}(n \log \log n + kC)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$

TABLE 2. Construction and modification time and space complexities.
[a]This assumes that either the number of sources and sinks is negligible (Bowe et al., 2012; Boucher et al., 2015), or the membership queries are not always exact (Li et al., 2016); otherwise, in the worst case, the space needed is $\Theta(nk)$.
[b]C is the number of connected components in the underlying undirected dBG.

(between $2k$ and $6k$, or less in case of a small connected component). A dictionary is used to store the k-mers associated with each root. Apart from these, no other node sequence is stored. The tree structure requires an additional cn bits to store, where c is implementation-dependent, and supports membership queries in $O(k)$ time. It is assumed that the space to store the root k-mers is a lower-order term of the whole structure, which is the case except when the graph consists of many small connected components (which is atypical for bioinformatics datasets).

To check for membership of a k-mer x, we start with the node x' which MPHF identifies as corresponding with x. We use the tree structure to follow x' up to its root (using at most $6k$ queries). Observe that if we are starting from a k-mer that is not present, due to how the tree structure is represented, we are not guaranteed to reach a k-mer in S after k (or even $6k$) steps: we may jump from one tree in the forest to another, not necessarily following a true path along the same tree. Thus, if a tree root cannot be reached after $6k$ steps, then we can conclude that $x' \neq x$ and hence $x \notin S$. Otherwise, we replay the steps that led to the root in the opposite

direction, using the root sequence together with dBG edge labels to reconstruct the sequence of x'. We then report x as being in S if and only if the reconstructed sequence is equal to x.

We summarize the query, construction, and modification time and space complexities of the key data structures in Tables 1 and 2. In the Appendix, we show how these complexities are derived for the cases when it is not explicit in the original papers.

6. Lower Bounds

How many bits are necessary to store S, in the worst case, so that membership queries can be answered (without mistakes)? Conway and Bromage (2011) provided an information theoretic answer, based on the fact that to store n elements from a universe of size U requires $\log \binom{U}{n}$ bits. In our case, we denote this lower bound by $L(n,k) = \log \binom{\sigma^k}{n}$ and, using standard inequality bounds, we have:

$$n \log(\sigma^k/n) \leq L(n,k) \leq n \log(\sigma^k/n) + n \log e$$

This asymptotically matches the space of Conway and Bromage's data structure (Table 2). The quantity $\log(\sigma^k/n)$ reflects the density of the set, and we have that $0 \leq \log(\sigma^k/n) \leq k \log \sigma$. If S is exponentially sparse, then $L(n,k) = \Theta(nk)$.

Chikhi et al. (2014) explored lower bounds for navigational data structures. Here, how many bits are necessary to store S, in the worst case, so that navigational (i.e. *fwd* and *bwd*) queries can be answered (without mistakes)? They showed that $L_{\text{nav}}(n,k) = 3.24n$ bits are required to represent a navigational data structure. Note that this beats the above lower bound for membership data structures, because a navigational data structure cannot answer arbitrary *memb* queries.

The above are traditional worst case lower bounds, meaning that, for any representation that uses less than $L(n,k)$ (respectively, $L_{\text{nav}}(n,k)$) bits for all possible sets S with n elements of k-mers, there will exist at least one input where the representation will produce a false answer to a membership (respectively, navigational) query. However, this is of limited interest in the bioinformatics setting, where the k-mers in S come from an underlying biological source. For example, the family of graphs used to prove the L_{nav} bound would never occur in bioinformatics practice. As a result, the value that worst-case lower bounds bring to practical representation of k-mer sets is limited. In fact, the static BOSS and the static Belazzougui data structures are able to beat this lower bound in practice by taking advantage of the fact that the dBG is typically highly connected.

The difficulty of finding an alternative to worst-case lower bounds is the difficulty of modeling the input distribution. Chikhi et al. (2014) considered the opposite end of the spectrum. They call S *linear* if the node-centric de Bruijn graph of S is a simple path (all internal nodes have in- and out-degree of 1). They showed that the number of bits needed to represent S that is linear is $L_{\text{lin}}(n,k) = 2n$. A linear k-mer set is in some sense the best case that can occur in practice. However, a linear k-mer set is much easier to represent than the sets arising in practice, hence L_{linear} it is too conservative of a lower bound.

An intermediate model was also considered by Chikhi et al. (2014), where S is parametrized by the number of simple paths in the de Bruijn graph. They used this parameter to describe how much space their representation takes, however, they did not pursue the interesting question of a lower bound parametrized by the number of simple paths.

7. Variations and extensions

There are natural variations and extensions of data structures for storing k-mer sets, which we describe in this section.

7.1. Membership of ℓ-mers for $\ell < k$.

A useful operation may be to check if S contains a given string u of length $|u| = \ell < k$. While it is usually easy to find if any k-mer begins with u, it may be the case that u does not appear as a prefix but still appears in S. One way to check for u's membership is to enumerate all the k-mers in S and then perform an exact string matching algorithm in $\mathcal{O}(nk)$ time (e.g. Knuth-Morris-Pratt). Another way is to attempt all $\sigma^{k-\ell}$ possible ways to complete a k-mer from u. Both these ways are prohibitively inefficient for most

applications. However, both the static BOSS and the FM-index on top of unitigs (Chikhi et al., 2014; Guo et al., 2018) data structures support checking u's membership in $\mathcal{O}(\ell)$ time; dynamic BOSS also supports this, in time $\mathcal{O}(\ell(1 + \log n/\log\log n))$.

7.2. **Variable-order de Bruijn graphs.** The *fwd* and *bwd* operations require an overlap of $k-1$ characters in order to navigate S. However, if such an overlap does not exist, it might make sense to look for a shorter overlap. The variable-order BOSS was introduced to allow this (Boucher et al., 2015). For a given K, it simultaneously represents all the dBGs for $k < K$. At any given time, the variable-order BOSS maintains an intermediate state, which is a value $k < K$ and a range of nodes (denoted as B) which share the same suffix of length k, representing a single node in the dBG for k. It supports new operations *shorter*() and *longer*() for changing the value of k (by one), running in $\mathcal{O}(\log K)$ and $\mathcal{O}(|B|\log K)$ time, respectively. The *bwd* operation runs in the same asymptotic time as BOSS, but *fwd* runs in $\mathcal{O}(\log K)$ time. A bidirectional variable order BOSS improved that *bwd* operation from $\mathcal{O}(K)$ to $\mathcal{O}(\log K)$ (Belazzougui et al., 2016b). The *memb* times are unaffected compared to BOSS. The space complexity is $n\log K + 4n + o(n)$ bits, adding an extra $n\log K$ bits to the space of BOSS.

7.3. **Double strandedness.** The *reverse complement* of a string is the string reversed and every nucleotide (i.e. character) replaced by its Watson-Crick complement. In bioinformatics applications, it is often useful to treat a k-mer and its reverse compliment as a single unit. There are two general ways in which data structures for storing k-mer sets can be adapted to achieve this.

The first way is to make all k-mers canonical. A k-mer is *canonical* if it is lexicographically no larger than its reverse complement. To make a k-mer x canonical, one replaces it by its reverse complement if x is not canonical. The elements of S are made canonical prior to construction of the data structure, and *memb* queries always make the k-mer canonical first. This approach works well in data structures that are hash-based (e.g. sorted list, hash table adjacency list, Conway and Bromage, Bloom filter) or the Bloom filter trie. The space of these data structures does not increase, but the query times increase by the $\mathcal{O}(k)$ operations that may be needed to make a k-mer canonical.

For a BWT-based data structure such as BOSS, using canonical k-mers is incompatible with the specialized *fwd* and *bwd* operations. In such cases, we can first modify S by checking, for every $x \in S$, if the reverse complement of x is in S, and, if not, adding this reverse complement to S. This increases the size of the data structure by up to a factor of two, but maintains the same time for *fwd* and *bwd* operations.

In case of unitig based representations, the unitigs themselves can be constructed on what is called a bidirected de Bruijn graph (Medvedev et al., 2007, 2019). A bidirected graph naturally captures the notion of double-stranded k-mer extensions in a graph-theoretic framework. The unitigs can then be indexed using their canonical form.

7.4. **Maintaining k-mer counts.** In many contexts it is natural to store a positive integer count associated with each k-mer in S. Alternatively, this may be viewed as storing a multi-set instead of a set. In the same way that a set of k-mers can be thought of as a de Bruijn graph, a multi-set of k-mers can be also thought of as a weighted de Bruijn graph.

Many of the data structures discussed naturally support maintaining counts, including operations to increment or decrement a count. Any of the data structures that associate some memory location with each k-mer in S can be augmented to store counts, e.g. a hash table adjacency list representation or a BOSS. More generally, if a data structure provides a method to obtain the rank of a k-mer within S (e.g. Conway and Bromage), that rank can be used as an index into an integer vector containing the counts. For Bloom filters, there also exist variants that allocate a fixed number of bits per k-mer to store the approximate counts (the counting Bloom filter, Fan et al. (2000)).

The downside of such representations, however, is that they are space inefficient when the distribution of count values is skewed. For example, in one typical situation, most k-mers will have a count of ≤ 10, but there will be a few with a count in the thousands. Since these representations

use a fixed number of bits to represent a count, they will waste a lot of bits for low count k-mers in order to support just a few k-mers with a large count. To alleviate this, variable-length counters can be used. Conway and Bromage (2011) proposed a tiered approach, storing higher order bits only as needed. More recently, the counting quotient filter (Pandey et al., 2017c) was designed with variable-length counters in mind; it was applied to store a k-mer multi-set by the Squeakr (Pandey et al., 2017d) and deBGR (Pandey et al., 2017b) algorithms.

Mäkinen et al. (2015, Section 9.7.2) also present a count-aware alternative to BOSS, also based on the BWT and following Välimäki and Rivals (2013). In this representation, a BWT is constructed without removing duplicate k-mers, and the count of a k-mer x can then be inferred by the number of entries in the BWT corresponding to x. This approach avoids storing an explicit count vector, however, it requires space to represent each extra copy of a k-mer. This trade-off can be beneficial when the count values are skewed and most k-mers have low counts.

7.5. **Sets of k-mer sets.** A natural extension of a k-mer set is a set of k-mer sets, i.e. $\{S_1, \ldots, S_m\}$, where each S_i is a k-mer set. Sets of k-mer sets have received significant recent interest as they are used to index large collections of sequencing datasets or genomes from a population. An equivalent way to think about this is a set of k-mers S where each k-mer x is associated with a set of genomes (often called colors) $c(x) \subseteq [m]$. A set of colors is referred to as a *color class*, and a color class j is said to *represent* a k-mer x if $c(x) = j$. If the underlying set of k-mers is intended to support navigational queries, then a representation of S is referred to as a *colored de Bruijn graph* (Iqbal et al., 2012). This is an extension of viewing a k-mer set as a de Bruijn graph to the case of multiple sets.

The literature has focused on two types of queries. The first is the basic k-mer color query: given a k-mer x, is $x \in S$, and, if yes, what is $c(x)$? The second is a color matching query: given a set of query k-mers Q and a threshold $0 < \Theta \leq 1$, identify all colors that contain at least a fraction Θ of the k-mers in Q.

Proposed representations have generally fallen into two categories. The first explicitly stores each k-mer's color class in a way that can be indexed by the k-mer. For example, Holley et al. (2016) proposed storing the color class of a k-mer at its corresponding leaf in a Bloom filter trie, while Pandey et al. (2017a) stored the color class in the k-mer's slot of a counting quotient filter. Alternatively, a BOSS can be used to store the k-mers and the colors can be stored in an auxiliary binary color matrix C (Muggli et al., 2017; Almodaresi et al., 2017). Here, $C[i, j] = 1$ if the i^{th} k-mer in the BOSS ordering has a color j. Instead of using a BOSS, k-mers in the color matrix can also be indexed using a minimal perfect hash function (Yu et al., 2018).

A column of the color matrix can be viewed as binary vector specifying the k-mer membership of S_i. A variation of this then replaces each column using a Bloom filter representation of S_i (Mustafa et al., 2018; Bradley et al., 2017). Thus, each row of the color matrix becomes a position in the Bloom filter, instead of a k-mer. This results in space savings, but representation of the color class is no longer guaranteed to be correct.

The color matrix can be compressed using a standard compression technique such as RRR or Elias-Fano encoding (Muggli et al., 2017). Further compression can be achieved based on the idea that, in some applications, many k-mers share the same color class. For example, Holley et al. (2016); Almodaresi et al. (2017); Pandey et al. (2017a) assign an integer code to each color class in increasing order of the number of k-mers it represents. Thus, frequently occurring color classes are represented using less bits. Yu et al. (2018) proposed an adaptive approach to encoding color classes. Based on how many colors a class represents, a class is stored as either a list of the colors, a delta-list encoding of the colors, or as a bitvector of length m. Finally, an alternative way to encode the color matrix based on wavelet trees is given by Mustafa et al. (2018).

The second category representations are based on the Bloofi (Crainiceanu and Lemire, 2015) data structure, which is designed to exploit that many S_is are similar and, more generally, many color classes have similar k-mer compositions. Here, each S_i is stored in a Bloom filter and a tree is constructed with each S_i as a leaf. Each internal node represents the union of the k-mers of its descendants, also represented as a Bloom filter. The Bloofi datastructure was adapted to the k-mer setting by Solomon and Kingsford (2016a), who called it the Sequence Bloom Tree. The

color matching query can be answered by traversing the tree top-down and pruning the search at any node where less than $\Theta|Q|$ k-mers match. Further improvements were made to reduce its size and query times (Harris and Medvedev, 2019; Sun et al., 2016; Solomon and Kingsford, 2016b). For example, k-mers that appear in all the nodes of a subtree can be marked as such to allow more pruning during queries, and the information about such k-mers can be stored at the root, thereby saving space (Sun et al., 2016; Solomon and Kingsford, 2016b). Using a hierarchical clustering to improve the topology of the tree also yields space savings and better query times (Sun et al., 2016). A better organization of the bitvectors was shown to reduce saturation and improve performance (Harris and Medvedev, 2019).

The first category of representations are designed with the basic k-mer color query in mind, though they can be adopted to answer the color matching query as well. The second category of methods, on the other hand, are specifically designed to answer the color matching query. They can be viewed as aggregating k-mer information at the color level, while the first category can be viewed as aggregating color information at the k-mer level.

8. Conclusion

In this paper, we have reviewed the various data structures used to represent k-mer sets. We hope that this area receives more systematic attention in the future, as k-mer set representations underly many bioinformatics tools. One promising avenue of research is to better and more explicitly model the distribution of k-mer sets that arise in sequencing data; such models can then uncover more efficient representations.

References

Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. 2017. Rainbowfish: A succinct colored de Bruijn graph representation. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 88. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. 2018. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics* 34, 13 (2018), i169–i177.

Jørgen Bang-Jensen and Gregory Z Gutin. 2008. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media.

Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. 2012. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology* 19, 5 (2012), 455–477.

Markus J Bauer, Anthony J Cox, and Giovanna Rosone. 2013. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science* 483 (2013), 134–148.

Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *European Symposium on Algorithms*. Springer, 682–693.

Djamal Belazzougui, Travis Gagie, Veli Mäkinen, and Marco Previtali. 2016a. Fully Dynamic de Bruijn Graphs. In *International Symposium on String Processing and Information Retrieval*. Springer, 145–152.

Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J Puglisi. 2016b. Bidirectional variable-order de Bruijn graphs. In *LATIN 2016: Theoretical Informatics*. Springer, 164–178.

Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637.

Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. 2015. Variable-Order de Bruijn Graphs. In *Data Compression Conference*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer (Eds.). IEEE Computer Society Press, 383–392.

Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. 2012. Succinct De Bruijn Graphs. In *Proceedings of the 12th International Conference on Algorithms in Bioinformatics (WABI'12)*. Springer-Verlag, Berlin, Heidelberg, 225–235.

Phelim Bradley, Henk den Bakker, Eduardo Rocha, Gil McVean, and Zamin Iqbal. 2017. Real-time search of all bacterial and viral genomic data. *bioRxiv* (2017), 234955.

Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. 2014. On the representation of de Bruijn graphs. In *International conference on Research in computational molecular biology*. Springer, 35–55.

Rayan Chikhi, Antoine Limasset, and Paul Medvedev. 2016. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics* 32, 12 (2016), i201–i208.

Rayan Chikhi and Guillaume Rizk. 2012. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In *WABI (Lecture Notes in Computer Science)*, Vol. 7534. Springer, 236–248.

Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 241–252.

Thomas C. Conway and Andrew J. Bromage. 2011. Succinct data structures for assembling large genomes. *Bioinformatics* 27, 4 (2011), 479.

Adina Crainiceanu and Daniel Lemire. 2015. Bloofi: Multidimensional bloom filters. *Information Systems* 54 (2015), 311–324.

Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. `https://doi.org/10.1145/321812.321820`

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)* 8, 3 (2000), 281–293.

Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Shan Muthukrishnan. 2009. Compressing and indexing labeled trees, with applications. *Journal of the ACM* 57, 1 (2009), 4:1–4:33.

Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *FOCS 2000: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 390–398.

Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2007. Compressed Representations of Sequences and Full-text Indexes. *ACM Trans. Algorithms* 3, 2, Article 20 (May 2007). `https://doi.org/10.1145/1240233.1240243`

Hongzhe Guo, Yilei Fu, Yan Gao, Junyi Li, Yadong Wang, and Bo Liu. 2018. deGSM: memory scalable construction of large scale de Bruijn Graph. *bioRxiv* (2018). `https://doi.org/10.1101/388454`

Robert S Harris and Paul Medvedev. 2019. Improved Representation of Sequence Bloom Trees. *bioRxiv* (2019), 501452.

Steffen Heinz, Justin Zobel, and Hugh E Williams. 2002. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)* 20, 2 (2002), 192–223.

Guillaume Holley, Roland Wittler, and Jens Stoye. 2016. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology* 11, 1 (2016), 3.

Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. 2012. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics* 44, 2 (2012), 226.

Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. 2016. ABySS 2.0: Resource-Efficient Assembly of Large Genomes using a Bloom Filter. *bioRxiv* (2016), 068338.

Dinghua Li, Ruibang Luo, Chi-Man Liu, Chi-Ming Leung, Hing-Fung Ting, Kunihiko Sadakane, Hiroshi Yamashita, and Tak-Wah Lam. 2016. MEGAHIT v1. 0: a fast and scalable metagenome assembler driven by advanced methodologies and community practices. *Methods* 102 (2016), 3–11.

Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 75. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. 2015. *Genome-scale algorithm design*. Cambridge University Press.

Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. 2005. An extension of the Burrows Wheeler transform to k words. In *Data Compression Conference, 2005. Proceedings. DCC 2005*. IEEE, 469.

Paul Medvedev, Rayan Chikhi, and Antoine Limasset. 2019. Bi-Directed Graphs in BCALM 2. `https://github.com/GATB/bcalm/blob/master/bidirected-graphs-in-bcalm2/bidirected-graphs-in-bcalm2` GitHub commit e9ba83c92999d9db03b71920b7354b213f545376.

Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. 2007. Computability of models for sequence assembly. In *International Workshop on Algorithms in Bioinformatics*. Springer, 289–301.

Michael Mitzenmacher. 2002. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)* 10, 5 (2002), 604–612.

Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. 2017. Succinct colored de Bruijn graphs. *Bioinformatics* 33, 20 (2017), 3181–3187.

Harun Mustafa, Ingo Schilken, Mikhail Karasikov, Carsten Eickhoff, Gunnar Rtsch, and Andr Kahles. 2018. Dynamic compression schemes for graph coloring. *Bioinformatics* (2018), bty632. `https://doi.org/10.1093/bioinformatics/bty632`

Gonzalo Navarro and Kunihiko Sadakane. 2014. Fully Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms* 10, 3, Article 16 (May 2014), 39 pages. `https://doi.org/10.1145/2601073`

Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 60–70.

Tony Pan, Rahul Nihalani, and Srinivas Aluru. 2018. Fast de Bruijn Graph Compaction in Distributed Memory Environments. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2018).

Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2017a. Mantis: A Fast, Small, and Exact Large-Scale Sequence Search Index. *bioRxiv* (2017), 217372.

Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017b. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, 14 (2017), i133–i141.

Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017c. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 775–787.

Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017d. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics* 34, 4 (2017), 568–575.

Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. 2012. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences* 109, 33 (2012), 13272–13277.

David Pellow, Darya Filippova, and Carl Kingsford. 2016. Improving Bloom Filter Performance on Sequence Data Using k-mer Bloom Filters. In *International Conference on Research in Computational Molecular Biology*. Springer, 137–151.

Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on*

Algorithms (TALG) 3, 4 (2007), 43.

Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. 2013. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In *Algorithms in Bioinformatics*, Aaron Darling and Jens Stoye (Eds.). Lecture Notes in Computer Science, Vol. 8126. Springer Berlin Heidelberg, 364–376. https://doi.org/10.1007/978-3-642-40453-5_28

Brad Solomon and Carl Kingsford. 2016a. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* (2016).

Brad Solomon and Carl Kingsford. 2016b. Improved Search of Large Transcriptomic Sequencing Databases Using Split Sequence Bloom Trees. *bioRxiv* (2016), 086561.

Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. 2016. AllSome Sequence Bloom Trees. *bioRxiv* (2016), 090464.

Niko Välimäki and Eric Rivals. 2013. Scalable and Versatile k-mer Indexing for High-Throughput Sequencing Data. In *Bioinformatics Research and Applications, 9th International Symposium, ISBRA 2013, Charlotte, NC, USA, May 20-22, 2013. Proceedings*. 237–248. https://doi.org/10.1007/978-3-642-38036-5_24

Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Chen Qian, and Jinze Liu. 2018. SeqOthello: Query over RNA-seq experiments at scale. *bioRxiv* (2018), 258772.

Appendix A. Derivations of complexities

A.1. **Conway and Bromage.** Conway and Bromage (2011) present separate structures for dense and sparse sets; in our case, the sparse bitmap representation (called sarray in Conway and Bromage (2011)) is relevant. The space taken by sarray is given in Table 1 of Conway and Bromage (2011) as $\mu \log \frac{\nu}{\mu} + 1.92\mu + o(\mu)$. In our case, $\mu = n$ and $\nu = \sigma^k$. Membership is implemented as a constant number of rank operations, which are supported in sarray in time $\mathcal{O}(\log \frac{\nu}{\mu}) + \mathcal{O}(\log^4 \mu / \log \nu)$ (Table 1 in Conway and Bromage (2011)). In terms of construction time, we did not find an analysis in either Conway and Bromage (2011) or Okanohara and Sadakane (2007). We show the construction time as $\Omega(nk)$, since it is at least necessary to hash each k-mer.

A.2. **Bloom filter tries.** The Bloom filter trie complexities depend on several internal parameters (e.g. ℓ, c, f, q, λ in the paper). For our analysis, we have treated these as constants, and, in particular, we have set $\ell = 1$ as it minimizes the complexity of operations. Yet, this is an extreme case that has not been explicitly considered in the original article, and Holley et al. (2016) suggested optimizations for performing faster extension queries that are not reflected by our analysis here. A more fine-grained analysis then we have done here is likely possible, in terms of these internal parameters.

A.3. **BOSS.** In Bowe et al. (2012), the time complexity of $memb(x)$ query (called $Index(x)$) is $\mathcal{O}(k(t_f + t_b(m, 2\sigma))$, where t_f is $\mathcal{O}(1)$ (rank & select (Raman et al., 2007)) for the static case and $\mathcal{O}(\log \sigma)$ (a balanced binary search tree) for the dynamic case, and t_b is the maximum of complexities of functions rank, select, and access on strings, which is $\mathcal{O}(\frac{\log \sigma}{\log \log n})$ for the static implementation (Ferragina et al., 2007) and $\mathcal{O}(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$ for the dynamic implementation (Navarro and Sadakane, 2014). Considering that the alphabet size is constant in our case, the static implementation makes $memb(x)$ query time complexity equal to $\mathcal{O}(k)$ and the dynamic complexity makes it $\mathcal{O}(k(1 + \frac{\log n}{\log \log n}))$.

The time complexity of $fwd(x, a)$ query (called $Outgoing(x, a)$) is $\mathcal{O}(t_f + t_b(m, 2\sigma))$, which is $\mathcal{O}(1)$ for the static case and $\mathcal{O}(\frac{\log n}{\log \log n})$ for the dynamic case. The time complexity of $bwd(x, a)$ query (called $Incoming(x, a)$) is $\mathcal{O}(k(t_f + t_b(m, 2\sigma)) \log \sigma)$, which is $\mathcal{O}(k \log \sigma)$ for the static case and $\mathcal{O}(k \log \sigma(1 + \frac{\log n}{\log \log n}))$ for the dynamic case. Both static (Ferragina et al., 2007) and dynamic (Navarro and Sadakane, 2014) rank & select implementations have the same asymptotic space complexity; therefore, both the static and dynamic BOSS have the same asymptotic space complexity.

A.4. **variable-order BOSS.** In the case of a constant alphabet, the variable-order BOSS (Boucher et al., 2015) representation uses the data structures of original BOSS and a new L^* array requiring $\mathcal{O}(n \log K)$ space (Boucher et al., 2015, Theorem 1). The $memb(x)$ query is used in the same way as in BOSS. Operations $fwd(x, a)$ and $bwd(x, a)$ for K-mers are also used in the same way as in BOSS. For k-mers with $k < K$ the operations are implemented in a different (slower) way: $fwd(x, a) = shorter(fwd(maxlen(x, a), a), k_v)$, $bwd(x) = shorter(bwd(maxlen(longer(x, k_v + 1), *)), k_v)$, $lastchar(x) = lastchar(maxlen(x, *))$. Note, $bwd(x)$ in the variable-order BOSS returns a list of nodes with an edge to x. In (Boucher et al., 2015, Section 5) variable-order BOSS $bwd(x)$ time complexity is $\mathcal{O}(\sigma(t_{bwd(x)} + \log k))$. Operation $maxlen([i, j], a)$ runs in $\mathcal{O}(\log |\Sigma|)$ time (i.e. $\mathcal{O}(1)$ time for $|\Sigma| = \text{const}$), $maxlen([i, j], *)$ runs in $\mathcal{O}(1)$ time. Operation $shorter([i, j], k)$ runs in time $\mathcal{O}(\log K)$ and operation $longer([i, j], k)$ runs in time $\mathcal{O}(|B| \log K)$, where B is a range of nodes sharing the same suffix of length k.

Rayan Chikhi, Center of Bioinformatics and Biostatistics and Integrative Biology - USR 3756, Institut Pasteur and CNRS, 25-28 rue du Docteur Roux, 75015 Paris, France
E-mail address: rayan.chikhi@pasteur.fr

Jan Holub, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 2700/9, 160 00, Prague 6, Czech Republic
E-mail address: Jan.Holub@fit.cvut.cz

Paul Medvedev, Department of Computer Science and Engineering and Department of Biochemistry and Molecular Biology, 506B Wartik Lab, University Park, The Pennsylvania State University, PA 16802, USA
E-mail address: pzm11@psu.edu