Data Structures to Represent a Set of k-long DNA Sequences

RAYAN CHIKHI, Center of Bioinformatics and Biostatistics and Integrative Biology
JAN HOLUB, Department of Theoretical Computer Science, Czech Technical University in Prague
PAUL MEDVEDEV, Center for Computational Biology and Bioinformatics

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 a k-mer set has emerged as a shared underlying component. A set of k-mers has unique features and applications that, over the past 10 years, have resulted in many specialized approaches for its representation. In this survey, we give a unified presentation and comparison of the data structures that have been proposed to store and query a k-mer set. We hope this survey will serve as a resource for researchers in the field as well as make the area more accessible to researchers outside the field.

CCS Concepts: • **Applied computing → Computational biology**; • **Theory of computation → Pattern matching**;

Additional Key Words and Phrases: k-mer sets, de Bruijn graphs, navigational data structures, Bloom filters, unitgs, FM-index, k-mers, biological sequencing data, data structures

ACM Reference format:
Rayan Chikhi, Jan Holub, and Paul Medvedev. 2021. Data Structures to Represent a Set of k-long DNA Sequences. *ACM Comput. Surv.* 54, 1, Article 17 (March 2021), 22 pages.
https://doi.org/10.1145/3445967

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.

Authors' addresses: R. Chikhi, Center of Bioinformatics and Biostatistics and Integrative Biology - USR 3756, Institut Pasteur and CNRS, 25-28 rue du Docteur Roux, 75015 Paris, France; email: rayan.chikhi@pasteur.fr; J. Holub, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 2700/9, 160 00, Prague 6, Czech Republic; email: Jan.Holub@fit.cvut.cz; P. 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; email: pzm11@psu.edu.

0360-0300/2021/03-ART17
https://doi.org/10.1145/3445967

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., anywhere between 50 characters and several thousands of characters, or more) 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, fixed-length substrings of the reads. These are called k-mers, where k refers to the length of the substring; equivalently, some papers use the term q-gram instead of k-mer. Such k-mer-based methods have become more popular in the past 10 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 a set 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.

In this survey, we describe published data structures for indexing a set of k-mers such that set membership can be checked either directly or by attempting to extend elements already in the set (called navigational queries, to be defined in Section 2). We evaluate the data structures based on their theoretical time for membership and navigational queries, space and time for construction, and time for insertion or deletion. We also describe known lower bounds on the space usage of such data structures and various extensions that go beyond membership and navigational queries. We do not describe the various applications of k-mer sets to biological problems, i.e., strategies for constructing the k-mer set from the biological data (e.g., sampling, error detection) or algorithms that use k-mer set data structures to solve some problem (e.g., assembly, genotyping). An example of an application we do not specifically discuss is the use of a k-mer set as an index, e.g., when a k-mer is used to retrieve a position in a reference genome.

Since these data structures are often developed in an applied context and published outside the theoretical computer science community, they do not consistently contain thorough mathematical analysis or even problem statements. There is the additional problem of inconsistent definitions and terminology. In this survey, we attempt to unify them under a common set of query operations, categorize them, and draw connections between them. We present a combination of: (1) non-specialized data structures (i.e., hash tables) that have been applied to k-mer sets as is, (2) non-specialized data structures that have been adapted for their use on k-mer sets, and (3) data structures that have been developed specifically for k-mer sets. We give a high-level overview of all categories, but we give a more detailed description for the third category. The survey can be read with only an undergraduate-level understanding of computer science, though knowledge of the FM-index would lead to a deeper understanding in some places.

Let S denote a set of n k-mers. Representing a set is a well-studied problem in computer science. However, the fact that the set consists of strings, and that the strings are fixed-length, lends structure that can be exploited for efficiency. There are other factors as well. First, in most applications, the alphabet has constant size, denoted by σ. Second, most applications revolve around sets where $n = o(\sigma^k)$; in this survey, we refer to these as *sparse* sets. 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 a k-mer set is what we call the *spectrum-like-property*. S has the *spectrum-like-property* if there exists a collection $\mathcal{G}$ of long strings that "generates" S. By "generates," we mean 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 $\mathcal{G}$. $\mathcal{G}$ is usually unknown. 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 to informally capture an important aspect of S in many applications that arise from sequencing. Our definition is necessarily imprecise to capture the huge diversity in how sequencing technologies are applied and how sequencing data is used. However, as we will show, this property is exploited by methods for representing a k-mer set and also drives the types of queries that are performed on them.

2 OPERATIONS

In this section, we describe a common set of operations that unifies many of the data structures for representing a set of k-mers. First, let us 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 supports are its construction and checking whether a k-mer x is in S (*memb*, which returns a Boolean value). 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 the 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 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. A forward extension of x is any k-mer y such that $y[1, k - 1] = x[2, k]$, and a backward extension is any k-mer y such that $y[2, k] = x[1, k - 1]$ (we use the notation $x[i, j]$ to refer to the substring of x starting from the ith character up to and including the jth character). Formally, given $x \in S$ and a character a, the $\mathit{fwd}(x, a)$ operation returns true if $x[2, k] \cdot a$ is in S (we use $\cdot$ to signify string concatenation). Similarly, the $\mathit{bwd}(x, a)$ operation checks whether $a \cdot x[1, k - 1]$ is in S. We refer to *fwd* and *bwd* operations as navigation operations.

We assume that a data structure maintains some kind of internal state corresponding to the last queried k-mer i.e., a $\mathit{memb}(x)$ query would leave the data structure in a state corresponding to x, a $\mathit{fwd}(x, a)$ query would leave the state corresponding to $x[2, k] \cdot a$, and so on. For example, for a hash table, the internal state after a $\mathit{memb}(x)$ query would correspond to the hash value of x and to the memory location of x's slot; in the case of an FM-index or a similar data structure, the internal state corresponds to an interval representing x.

We also assume that prior to a call to $\mathit{fwd}(x, a)$ or $\mathit{bwd}(x, a)$, the data structure is in a state corresponding to x. In this way, $\mathit{fwd}(x, a)$ and $\mathit{bwd}(x, a)$ are different from $\mathit{memb}(x[2, k] \cdot a)$ and $\mathit{memb}(a \cdot x[1, k - 1])$, respectively. For example, it would be invalid to execute $\mathit{fwd}(ACG, T)$ after executing $\mathit{memb}(CCC)$, because the *memb* operation would leave the data structure in a state corresponding to *CCC* and executing *fwd* requires it to be in a state corresponding to *ACG*. For data structures that do not support *fwd* or *bwd* explicitly or do not maintain an internal state, there is always the default implementation using the corresponding membership query.

In the following, we will first summarize some basic data structures for the above problem (Section 3). In Section 4, we will make the connection to de Bruijn graphs and present data structures that aim for fast *fwd* and *bwd* queries. In Section 5, we present special type of data structures where *memb* queries are very expensive or impossible, but navigational queries are cheap. 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. We then continue to other aspects. In Section 6, we describe the known space lower bounds for storing a set of k-mers. Finally, in Section 7, we describe various variations on and extensions of the data structures presented in Sections 3–5.

We note that the definition of S as a set implies that there is no count information associated with a k-mer in S. However, some of the data structures we will present also support maintaining count information with each k-mer. Rather than present how this is done together with each data structure that supports it, we have a separate section (Section 7.4) dedicated to how the presented data structures can be adapted to store count information.

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 any 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, as it is dominated by other approaches we will discuss (e.g., unitig-based approaches or BOSS). But it can be used by someone with very limited computer science background, making it still relevant.

Sorted lists can be partitioned to speed up queries. In this approach, taken by Wood and Salzberg [2014], the k-mers are partitioned according to a minimizer function. For a given $\ell < k$, an ℓ-minimizer of a k-mer x is the smallest (according to some given permutation function) ℓ-mer substring of x [Roberts et al. 2004; Schleimer et al. 2003]. A minimizer function is a function that maps a k-mer to its minimizer, or, equivalently, to an integer in $\{1, \ldots, \sigma^\ell\}$. In the partitioned sorted list approach, the k-mers within each partition are stored in a separate sorted list, and a separate direct-access table maps each partition to the location of the stored list. For this table to fit into memory, ℓ should be small (e.g., $\ell = 13$ for $\sigma = 4$). This approach can work well to speed up queries when there are not many k-mers in each partition. However, the space used is still $\Theta(nk)$, which is inefficient compared to more recent methods we will present.

Two traditional types of data structures to represent sets of elements are binary search trees and hash tables. A binary search tree and its variants require $O(\log n)$ time for membership queries and are in most aspects worse than a string trie [Mäkinen et al. 2015]. To the best of our knowledge, binary search trees have not been used for directly indexing k-mers. In a **hash table**, the amortized time for a membership query, insertion, deletion, *fwd* and *bwd* is equivalent to the time for hashing a k-mer [Cormen et al. 2009]. Hashing a k-mer generally requires $O(k)$ time, but one can also use rolling hash functions. In a rolling hash function [Lemire and Kaser 2010], if we know the hash value for a k-mer x, we can compute the hash value of any forward or backward extension of x in $O(1)$ time. Using a rolling hash function can therefore improve the *fwd*/*bwd* query time to $O(1)$. These fast query and modification times and the availability of efficient and easy-to-use hash table libraries in most popular programming languages make hash tables 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 [2011] were of the first to consider more compact representations of a k-mer set. 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. The field of compact data-structures [Navarro 2016] concerns exactly with how to store such bitvectors space-efficiently. In this case, 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 $n = O(\sigma^{k(1-\epsilon)})$), then the space needed is $\Omega(nk)$.

3.1 Approximate Membership Query Data Structures

An approximate membership query data structure is a type of probabilistic data structure that represents a set in a space-efficient manner 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. Note that approximate membership queries are not related to the type of queries that ask whether S contains a k-mer with some bounded number of mismatches (e.g., one substitution) to the query k-mer.

Bloom filters [Bloom 1970] (abbreviated BF) are a classical example of an approximate membership data structure that has found widespread use in representing a k-mer set (see Broder and Mitzenmacher [2003] for a definition and analysis of Bloom filters). Some of the earliest applications were by Shi et al. [2010] and Stranneheim et al. [2010]. BFs applied to k-mers support *insert*, *memb*, *fwd*, and *bwd* operations in the time it takes to hash a k-mer (usually $\Theta(k)$, except for rolling hash functions) and take $O(n)$ space. A BF does not support *delete*(x), though there are variants of BFs that make tradeoffs to support it in $\Theta(k)$ time (e.g., counting BFs [Fan et al. 2000] and spectral BFs [Cohen and Matias 2003]). Further time-space tradeoffs can be achieved by compressing a BF using RRR [Raman et al. 2007] encoding [Mitzenmacher 2002]. See Tarkoma et al. [2011] for a survey of BF variations and the tradeoffs they offer.

Pellow et al. [2017] developed several modifications of a Bloom filter, specifically for a k-mer set. They take advantage of the spectrum-like property to either reduce the false positive rate or decrease the space usage. The general idea is that when S has the spectrum-like property, most of its k-mers will have some backward and forward extension present in S. The (hopefully small amount of) k-mers for which this is not true are maintained in a separate hash table. For the rest, to determine whether a k-mer x is in S, they make sure the BF contains not only x but also at least one forward and one backward extension. Using similar ideas, they give other versions of a BF for when S is the spectrum of a read set or of one string. In another paper, Chu et al. [2018] developed what they called a multi-index BF, which similarly takes advantage of the spectrum-like property (details omitted).

Bloom filters are popular, because they reduce the space usage to $O(n)$ while maintaining $O(k)$ membership query time. BFs and their variants are also valuable for their simplicity and flexibility. However, operations on Bloom filters generally require access to distant parts of the data structure and therefore do not scale well when they do not fit into RAM. Here, we highlight some more advanced approximate membership data structures that offer better performance and have been applied to k-mers sets. There is the quotient filter [Bender et al. 2012] and the counting quotient filter [Pandey et al. 2017a], which have been applied to storing a k-mer set in Pandey et al. [2017b] and Pandey et al. [2018]. There is also the quasi-dictionary [Marchet et al. 2018] and ℓ-Othello [Liu et al. 2018], both generally applicable to any set of elements but applied to a k-mer set by the authors. Cuckoo filters [Fan et al. 2014] are another approximate membership data structure that has been applied to k-mers [Zentgraf et al. 2020].

3.2 String-based Indices

There is a rich literature of string-based indices [Mäkinen et al. 2015], some of which can be modified to store and query a k-mer set. One of the most popular string-based indices to be applied to

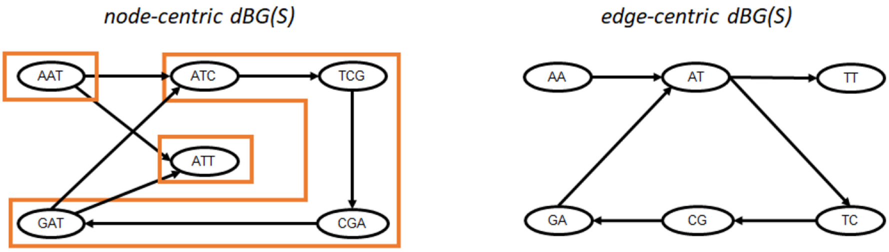

Fig. 1. An example of the node-centric de Bruijn graph (left) and the edge-centric one (right). Both graphs are built for $S = \{AAT, ATC, TCG, CGA, GAT, ATT\}$. There are three maximal unitigs in the node-centric graph, highlighted in the figure with orange rectangles. The spellings of the three maximal unitigs are AAT, $ATCGAT$, and ATT.

bioinformatics is the **FM-index**[1] [Ferragina and Manzini 2000]. It 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 (by treating every k-mer in S as a separate string), though we are unaware of such an application in practice. In theory, it results in $O(nk)$ construction time and $O(k)$ *memb* query time [Bauer et al. 2013]. A naive implementation of *fwd* and *bwd* operations in this setting would require a new *memb* query; however, we hypothesize that a more sophisticated approach, using bidirectional indices, may improve the runtime (this, however, does not appear in the literature and is not proven). However, the FM-index is not usually directly used for storing k-mers; rather, it is either used in combination with other strategies (e.g., DBGFM and deGSM, which we will describe in Section 4.2) or in a form specifically adapted to k-mer queries (i.e., the BOSS structure, which we will describe in Section 4.1).

Another popular string-based index is the trie data structure and its variations. A trie is a tree-based index known for its fast query time, with strings labeling nodes and/or edges (see Mäkinen et al. [2015] for details). Tries have been adopted to the k-mer set setting in a data structure called 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 ith part is then stored within a node at the ith 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 ($O(k)$) but requires $O(nk)$ space.

4 DE BRUIJN GRAPHS

A de Bruijn graph provides a useful way to think about a k-mer set that has the spectrum-like-property and for which *fwd* and *bwd* operations should be supported more efficiently than membership operations. A de Bruijn graph (dBG) is directed graph built from a set of k-mers S. In the *node-centric* dBG, the node set is given by S and there is an edge from x to y iff the last $k-1$ characters of x are equal to the first $k-1$ characters of y. In a *edge-centric* dBG, 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 $x[2, k]$. In other words, the k-mers of S are nodes in the node-centric dBG and edges in the edge-centric dBG. Figure 1 shows an example. The graphs represent equivalent information. Technically, the

[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).

node-centric dBG of S is a line graph [Bang-Jensen and Gutin 2009] of the edge-centric dBG of S, and without loss of generality, we mostly focus our discussion on node-centric dBGs.

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. It found its initial application in bioinformatics in genome assembly algorithms [Simpson and Pop 2015]. We do not discuss this application here, but rather, we discuss its relationship to the representation of a k-mer set.

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 a k-mer set. 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 formalism of de Bruijn graphs 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 article, in- and out-edge queries are equivalent to *bwd* and *fwd* queries, respectively. In particular, 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 $x[2,k] \cdot a$.

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. Concretely, each node u potentially has 2σ outgoing edges, corresponding to the possible σ forward extensions and σ backward extensions. Thus, we can use one bit for each of the 2σ possible edges to indicate their presence/absence. The navigational operations still require the time needed to hash a k-mer, because the hash value for the extension needs to be calculated 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 (relative no arbitrary graphs) 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 the edges' extension characters (i.e., for each edge x, the character $x[k]$), sorted by the concatenation of the reverse of the source node label and the extension character (i.e., $x[k-1] \cdot x[k-2] \cdots x[1] \cdot x[k]$). The details of the query algorithm are too involved to present here, and we refer the reader to either the original paper or to Mäkinen et al. [2015]. 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 $O(k)$ 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 $5n + o(n)$ (this is due to a distinct separator character being needed, as described in Bowe et al. [2012]). Otherwise, in the worst case, the space needed becomes $\Theta(nk)$ [Boucher et al. 2015; Bowe et al. 2012]. In the version given by Li et al. [2016], the space is always $6n + O(1)$, but then membership queries sometimes give incorrect answers. BOSS achieves a $O(1)$ runtime for the *fwd* operation, while *bwd* still runs in $O(k)$ time. The *bwd* query time can further be reduced to $O(1)$ using the method of Belazzougui et al. [2016b], at the cost of $O(n)$ extra space. This representation is static, but a dynamic one is also possible by sacrificing some query time [Belazzougui et al. 2016b; Bowe et al. 2012]. Like approximate membership data structures, BOSS achieves $O(n)$ space and $O(k)$ *memb* query time. The main difference is that approximate data structures have false positives while BOSS only achieves the $O(n)$ space when the number of sources/sinks is small.

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]). See Figure 1 for an example of maximal unitigs. 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 $t = x_1 x_2[k] \cdots x_\ell[k]$ with the property that a k-mer x is a substring of t 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 [Guo et al. 2019; Pan et al. 2020]; more generally, algorithms for compaction are often presented as part of genome assembly algorithms, which are too numerous to cite here.

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 to allow *memb* queries. In **deGSM** [Guo et al. 2019], the authors similarly build a BWT (which is the major component of an FM-index) of the maximal unitigs; but, they demonstrate how this can be done more efficiently by not explicitly constructing the strings of maximal unitigs (details omitted). These representations allow for $O(k)$ *memb* 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.

Another approach to index unitigs is taken by **Bifrost** [Holley and Melsted 2019], using minimizers. Bifrost builds a hash table where the keys are all the distinct minimizers of S and the values of the locations of those minimizers in the maximal unitigs. The membership of a k-mer is then checked by first computing its minimizer and then checking all the minimizer occurrences in the unitigs for a full match. The index is dynamic, i.e., it intelligently recomputes the unitigs and the minimizer index based on a k-mer insertion or deletion.

Before presenting other unitig-based indices, we make an aside to introduce minimal perfect hash functions. 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 $n - 1$. Minimal perfect hash functions (MPHF) can in theory be efficiently constructed and evaluated; we omit the details and refer the reader to Belazzougui et al. [2009] for an example. When applied to a k-mer set S, one can construct an MPHF in $O(nk)$ 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 $O(k)$ time. There exists an efficient implementation of MPHF for a k-mer set, BBHash [Limasset et al. 2017], designed to handle sets of billions of k-mers. The advantage of an 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 MPHF value of a k-mer as its index into the array. Unlike a hash table, this requires $O(n)$ instead of $O(nk)$ space. The disadvantage of an MPHF is that if it is given a k-mer $x \notin S$, then 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 article.

The **pufferfish** index [Almodaresi et al. 2018] uses an 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. A similar approach is the **BLight** index [Marchet et al. 2019b]. It also uses an MPHF to map k-mers to locations in unitigs, though it does it in a somewhat different way (we omit the details here).

Břinda [2016], Břinda et al. [2020], and Rahman and Medvedev [2020] recently extended the idea of unitig-based representations to **spectrum-preserving string set** representations (alternatively, these are referred to as **simplitigs**). They observed that what makes unitigs useful as a representation is that they contain the exact same k-mers as S, without any duplicates. They defined a spectrum-preserving string set representation as any set of strings that has this property and gave a greedy algorithm to construct one. The resulting simplitigs had a substantially lower number of characters than unitigs in practice. To support *memb* queries, simplitigs were combined with an FM-index [Rahman and Medvedev 2020] in the same manner that unitigs were combined with an FM-index to obtain DBGFM.

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 access pattern to reduce its space usage, as we will see in this section. Formally, a *navigational data structure* is one where *memb* queries are either very expensive or impossible, but *fwd* and *bwd* queries are cheap (e.g., $O(k)$). 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, as follows. 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. Given x's hash value, it takes only $O(1)$ time to find out if an extension exists, but the queries take $O(k)$ 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 $O(1)$ 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 an MPHF can be used as described above. This approach of indexing the extensions was taken by Limasset et al. [2016]. 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 [Jackman et al. 2017; Salikhov et al. 2013], 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 ($i > 1$) stores the k-mers that are false positives of B_{i-1}. BF-based navigational data structures support exact *fwd*/*bwd* queries in $O(k)$ time (or $O(1)$ 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 navigational queries are exact.

Belazzougui et al. [2016a] proposed a mechanism to transform their navigational data structure (described earlier in this section) into a membership data structure. They give both a static and dynamic version; we present the static one here. They first find a forest of node-disjoint rooted trees that is a node-covering subgraph of the dBG. Each tree has bounded height (between $2k$ and $6k$, or less in case of a small connected component). They build an MPHF of S and use it to store the adjacency list of the dBG, as described above. They also use it to record, for each k-mer, whether it is a root in the forest and in case it is not, a number between 0 and $2|\Sigma|$ to represent which navigational query will lead to its parent. A dictionary is used to store the node sequences of 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.

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 stored navigation instructions to follow x' up to its root (using at most $6k$ queries). If a tree root cannot be reached after $6k$ steps, or if any of the navigational instructions violate the information in the MPHF adjacency list, then we can conclude that $x' \neq x$ and hence $x \notin S$. If a tree root is reached within $6k$ steps, then $x \in S$ if and only if the sequence of the root (computed dynamically from traveling up the tree) is equal to the stored k-mer associated with the root.

6 SPACE LOWER BOUNDS

How many bits are necessary to store S, in the worst case, so 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 navigational 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 (for $\sigma = 4$). 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 a k-mer set 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 a de Bruijn graph that 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 single unitig. 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} 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 maximal unitigs in the de Bruijn graph. They used this parameter to describe how

Table 1. Query Complexities

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)	$k(1 + \frac{\log n}{\log \log n})$	$\frac{\log n}{\log \log n}$	$k(1 + \frac{\log n}{\log \log n})$
unitig-based[2]	k	[c]1 or k	[c]1 or k
Belazzougui et al[3]	k	[a]1	[a]1

Big O notation is implied for all the complexities, but the O symbol is omitted from the table for clarity.

[a] There is no specialized navigational query, so the time is the same as for *memb*.

[b] $O(1)$ occurs if a rolling hash function is used, otherwise there is no specialized navigational query.

[c] For DBGFM and deGSM, $O(1)$ holds if the extension lies on the same unitig; for BLight, it holds if the extension lies on the same super-k-mer; for pufferfish, it holds 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. 2019], pufferfish [Almodaresi et al. 2018], and BLight [Marchet et al. 2019b].

[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.

much space their representation takes, however, they did not pursue the interesting question of a lower bound parametrized by the number of maximal unitigs.

An alternative to traditional worst-case lower bounds or modeling the input distribution is to derive more instance-specific lower bounds. Typically, a lower bound is derived as a function of the input size, but a more instance-specific lower bound might be a function of the degree distribution of the de Bruijn graph or something even more specific to the graph structure. These types of lower bounds are extremely satisfying when they can be used to show an algorithm is instance-optimal, i.e., it matches the lower bound on every instance. Rahman and Medvedev [2020] derive such a lower bound for the number of characters in a spectrum-preserving string set representation. Their lower bound did not match the performance of their greedy algorithm in the worst case, but it came very close (within a factor of 2%) on the evaluated input.

7 VARIATIONS AND EXTENSIONS

There are natural variations and extensions of data structures for storing a k-mer set, which we describe in this section. These are not included in Tables 1 and 2, because they do not neatly fit into the framework of those tables.

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 $|u| = \ell < k$. In some data structures, like the Bloom filter trie, it is easy to find if a k-mer begins with u, but there is no specialized way to check if u appears as a non-prefix 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 $O(nk)$ time (e.g., Knuth-Morris-Pratt, described in the textbook of Cormen et al. [2009]). 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. 2019] data structures support checking u's membership in $O(\ell)$

Table 2. Construction and Modification Time and Space Complexities

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

Construction space refers to the size of the constructed data structure, rather than to the memory used by the construction algorithm.

[a] This assumes that either the number of sources and sinks is negligible [Boucher et al. 2015; Bowe et al. 2012], 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.

time; dynamic BOSS also supports this, in time $O(\ell(1 + \log n / \log \log n))$. We omit the details of these implementation here.

7.2 Variable-order de Bruijn Graphs

The *fwd* and *bwd* operations require an overlap of $k - 1$ characters to navigate S. However, if such an overlap does not exist, then in some applications it makes 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 $k < K$, as follows: 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) that 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 $O(\log K)$ and $O(|B| \log K)$ time, respectively. The *bwd* operation runs in the same asymptotic time as BOSS, but *fwd* runs in $O(\log K)$ time. A bidirectional variable order BOSS improved that *bwd* operation from $O(K)$ to $O(\log K)$ [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 many applications, it is often useful to treat a k-mer and its reverse complement as being identical. There are two general ways in which data structures for storing a k-mer set 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 $O(k)$ operations that may be needed to make a k-mer canonical.

For a data structure such as BOSS, using canonical k-mers is incompatible with the specialized *fwd* and *bwd* operations. For such cases, there is a second way to handle reverse complements. Concretely, we can compute the reverse complement closure of S, as follows: We 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. 2019, 2007]. 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. We omit the details here.

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, a BOSS, or a representation based on unitigs or on a spectrum-preserving string set. 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 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. 2017a] was designed with variable-length counters in mind; it was applied to store a k-mer multi-set by the Squeakr [Pandey et al. 2017b] and deBGR [Pandey et al. 2017c] 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 tradeoff 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) $c(x) \subseteq \{1 \ldots m\}$. A set of colors is referred to as a *color class*. 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. [2018] 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 [Almodaresi et al. 2017; Muggli et al. 2017]. Here, $C[i, j] = 1$ if the ith 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] or a unitig-based representation [Holley and Melsted 2019].

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 [Bingmann et al. 2019; Bradley et al. 2019; Mustafa et al. 2019]. 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 is sometimes compressed using a standard compression technique such as RRR [Raman et al. 2007] 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], and Pandey et al. [2018] assign an integer code to each color class in increasing order of the number of k-mers that belong to it. 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 color class contains, the class is stored as either a list of the colors, a delta-list encoding of the colors, or as a bitvector of length m. Almodaresi et al. [2019] take advantage of the fact that adjacent k-mers in the de Bruijn graph are likely to have similar color classes; they then store many of the color classes not as an explicit encoding but as a difference vector to a similar color class. Finally, an alternative way to encode the color matrix based on wavelet trees is given by Mustafa et al. [2019].

The second category of representations are based on the Bloofi [Crainiceanu and Lemire 2015] data structure, which is designed to exploit the fact 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 [2016], 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 2020; Solomon and Kingsford 2018; Sun et al. 2018]. 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 [Solomon and Kingsford 2018; Sun et al. 2018]. Using a hierarchical clustering to improve the topology of the tree also yields space savings and better query times [Sun et al. 2018]. A better organization of the bitvectors was shown to reduce saturation and improve performance [Harris and Medvedev 2020].

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, however, 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. For a more thorough survey of this topic, please see Marchet et al. [2019a].

8 CONCLUSION

In this article, we have surveyed data structures for storing a DNA k-mer set in a way that can efficiently support membership and/or navigational queries. This problem falls into the more general category of indexing a set of elements, which has been widely studied in computer science. The aspects of a DNA k-mer set that make it unique are that the elements are fixed length strings over a constant-sized alphabet, the set is sparse, and k is much less than n. A DNA k-mer set tends to also have what we have termed the spectrum-like-property. This property is hard to capture with mathematical precision, but it has been a major driver behind the design of specialized data structures. Another way that a DNA k-mer set is different from a general set is that queries are sometimes more constrained than arbitrary membership queries. In particular, navigational queries start from a k-mer that is known to be in the set and ask which of its extensions are also present.

We now give a summary of the major developments in this field. Some methods for storing a set proved to be useful right out-of-the-box, with the major examples being hash tables, Bloom filters, and sparse bitvectors. These methods are generic, in the sense that there is nothing specific to k-mer sets about them. Hash tables and Bloom filters, especially, gained widespread use because of their broad software availability and conceptual simplicity, respectively. These two offered a tradeoff between query accuracy and space; concretely, Bloom filters require only $O(n)$ space but have false positives, while hash tables have no false positives but require $O(nk)$ space. They both offered fast query times of $O(k)$ for membership and $O(1)$ for navigational queries (assuming rolling hash functions are used). Beyond these, other generic methods found applicability in k-mer sets, especially approximate membership query data structures. These offer both practical and theoretical improvements; however, describing these requires a more fine-grained analysis than we are able to provide here.

Generic data structures were also modified to take advantage of properties inherent to a DNA k-mer set, either simply that the strings are of fixed length or, more strongly, have the spectrum-like-property. The most notable examples of this were the works by Pellow et al. [2017] to modify Bloom filters, by Holley et al. [2016] to modify string tries (i.e., the Bloom filter trie data structure), and by Bowe et al. [2012] to modify the FM-index (i.e., BOSS data structure). Pellow et al. [2017] improved the space usage of Bloom filters, though the theoretical analysis is beyond the scope of this survey. The improvements of Holley et al. [2016] to a string trie were more practical and difficult to theoretically analyze. Bowe et al. [2012] were able to simultaneously achieve the $O(n)$ space usage of Bloom filters and the perfect accuracy of a hash table without affecting the query times. This, however, does not hold in the worst case, because it assumes that the number of sources and sinks in the de Bruijn graph is negligible. Later papers showed how to modify BOSS to achieve different tradeoffs [Belazzougui et al. 2016b; Li et al. 2016].

There were also two novel types of data structures developed specifically for the k-mer setting. The first was unitig-based representations, proposed by Chikhi et al. [2014] and later extended to spectrum-preserving string set representations by Břinda [2016], Rahman and Medvedev [2020], and Břinda et al. [2020]. These representations work by first constructing the unitigs and then building an index on top of them. The type of index varies: The FM-index is used by Chikhi et al. [2014] and Guo et al. [2019], while a minimum perfect hash function is used by Almodaresi et al. [2018], Marchet et al. [2019b], and Holley and Melsted [2019]. Unitig-based representations were specifically designed to exploit the spectrum-like-property to save space, resulting in $O(n + U(k - 1))$ space (U is the number of maximal unitigs in the input). Membership and navigation remain efficient ($O(k)$ and $O(1)$, respectively), except that for k-mers at the boundaries of unitigs, navigation takes $O(k)$. The idea is that in practice, the spectrum-like-property implies that U is much smaller than n, resulting in low space and making boundary k-mers rare in practice. A direct comparison between unitig-based representations and other representations (e.g., BOSS) to determine the regimes in which one outperforms the other has not, to the best of our knowledge, been attempted; this includes either a theoretical or a comprehensive empirical analysis.

The second type of data structure developed specifically for a k-mer set is a navigational data structure, which exploits the way that a DNA k-mer set is often queried. These data structures retain $O(1)$ navigational queries but sacrifice the efficiency and/or feasibility of membership queries to achieve $O(n)$ space. Chikhi and Rizk [2012] were the first to use such a data structure, and Chikhi et al. [2014] later formalized the idea; other navigational data structures were later developed by Bankevich et al. [2012], Salikhov et al. [2013], Jackman et al. [2017], Belazzougui et al. [2016a], and Limasset et al. [2016].

Reading through the literature in this field, one often encounters papers on the representation of de Bruijn graphs as opposed to representation of a k-mer set. The distinction between the two is unclear to us, as a de Bruijn graph and a k-mer set represent equivalent information (i.e., there is a bijection between the universe of k-mer sets and the universe of de Bruijn graphs). One distinction may be that the term "de Bruijn graph" implies that edge queries (which in the node-centric version correspond to navigational queries, in our terminology) are efficient, while the term "k-mer set" does not connote anything about navigation. However, "de Bruijn graph" obfuscates the fact that there are no degrees of freedom in defining the edge set: Once the node labels (i.e., k-mers) are determined, so are the edges. This is in the node-centric setting, but in the edge-centric setting, it is the nodes that are determined once the edge labels (i.e., k-mers) are fixed.

Beyond the data structures, we also discussed what is known about space lower bounds. Unfortunately, there have been only limited results. Besides the basic information-theoretic lower bound by Conway and Bromage [2011], nothing is known for membership data structures. For navigational data structures, Chikhi et al. [2014] provided some lower bounds; however, these are of limited practical use, because they only consider worst-case lower bounds, which are easily beat on real data. Within the confines of spectrum-preserving string set representations, instance-specific lower bounds were successfully applied empirically to demonstrate the near-optimality of the greedy representation on real data.

In this survey, we did not discuss in any detail how DNA k-mer sets are used in practice; we assume that there is some algorithm that takes a set of reads and extracts a k-mer set from them in a way that is useful to downstream algorithms. However, bringing such algorithms into some kind of unified framework would be a fascinating topic for another survey.

We hope that this area receives more systematic attention in the future, as k-mer set representations underly many bioinformatics tools. This might include expanding the set of operations beyond what we have described here to better capture the way a DNA k-mer set is used. Another 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 as well as provide useful lower bounds. Progress in the field can also come through the creation of benchmarking datasets and through impartial competitive assessment of existing tools (e.g., as in Bradnam et al. [2013]; Sczyrba et al. [2017]). The ultimate goal, though, remains practical: to come up with data structures that improve space and query time of existing ones.

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 $O(\log \frac{\nu}{\mu}) + O(\log^4 \mu / \log \nu)$ (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 article). 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 navigational queries that are not reflected by our analysis here. A more fine-grained analysis than 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 $O(k(t_f + t_b(m, 2\sigma))$, where t_f is $O(1)$ (rank & select [Raman et al. 2007]) for the static case and $O(\log \sigma)$ (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 $O(\frac{\log \sigma}{\log \log n})$ for the static implementation [Ferragina et al. 2007] and $O(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$ 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 $O(k)$ and the dynamic complexity makes it $O(k(1 + \frac{\log n}{\log \log n}))$.

The time complexity of $fwd(x, a)$ query (called $Outgoing(x, a)$) is $O(t_f + t_b(m, 2\sigma))$, which is $O(1)$ for the static case and $O(\frac{\log n}{\log \log n})$ for the dynamic case. The time complexity of $bwd(x, a)$ query (called $Incoming(x, a)$) is $O(k(t_f + t_b(m, 2\sigma)) \log \sigma)$, which is $O(k \log \sigma)$ for the static case and $O(k \log \sigma (1 + \frac{\log n}{\log \log n}))$ 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 $O(n \log K)$ 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 $O(\sigma(t_{bwd(x)} + \log k))$. Operation $maxlen([i, j], a)$ runs in $O(\log |\Sigma|)$ time (i.e., $O(1)$ time for $|\Sigma| = \text{const}$), $maxlen([i, j], *)$ runs in $O(1)$ time. Operation $shorter([i, j], k)$ runs in time $O(\log K)$, and operation $longer([i, j], k)$ runs in time $O(|B| \log K)$, where B is a range of nodes sharing the same suffix of length k.

REFERENCES

Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2019. An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. In *Proceedings of the International Conference on Research in Computational Molecular Biology (Lecture Notes in Computer Science)*, Vol. 11467. Springer, 1–18. DOI: https://doi.org/10.1007/978-3-030-17083-7_1

Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. 2017. Rainbowfish: A succinct colored de Bruijn graph representation. In *WABI 2017: Algorithms in Bioinformatics (LIPIcs-Leibniz International Proceedings in Informatics)*, Russell Schwartz and Knut Reinert (Eds.), Vol. 88. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 18:1–18:15. DOI: https://doi.org/10.4230/LIPIcs.WABI.2017.18

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. DOI: https://doi.org/10.1093/bioinformatics/bty292

Jørgen Bang-Jensen and Gregory Z. Gutin. 2009. *Digraphs: Theory, Algorithms and Applications*. Springer Science & Business Media. DOI: https://doi.org/10.1007/978-1-84800-998-1

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. *J. Comput. Biol.* 19, 5 (2012), 455–477. DOI: https://doi.org/10.1089/cmb.2012.0021

Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. 2013. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.* 483 (2013), 134–148. DOI: https://doi.org/10.1016/j.tcs.2012.02.002

Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *ESA 2009: European Symposium on Algorithms (Lecture Notes in Computer Science)*, Vol. 5757. Springer, 682–693. DOI: https://doi.org/10.1007/978-3-642-04128-0_61

Djamal Belazzougui, Travis Gagie, Veli Mäkinen, and Marco Previtali. 2016a. Fully dynamic de Bruijn graphs. In *SPIRE 2016: String Processing and Information Retrieval (Lecture Notes in Computer Science)*, Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai (Eds.), Vol. 9954. Springer, 145–152. DOI: https://doi.org/10.1007/978-3-319-46049-9_14

Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. 2016b. Bidirectional variable-order de Bruijn graphs. In *LATIN 2016: Theoretical Informatics (Lecture Notes in Computer Science)*, Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez (Eds.), Vol. 9644. Springer, 164–178. DOI: https://doi.org/10.1007/978-3-662-49529-2_13

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. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637. DOI: https://doi.org/10.14778/2350229.2350275

Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. 2019. COBS: A compact bit-sliced signature index. In *SPIRE 2019: String Processing and Information Retrieval (Lecture Notes in Computer Science)*, Vol. 11811. Springer, 285–303. DOI: https://doi.org/10.1007/978-3-030-32686-9_21

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426. DOI: https://doi.org/10.1145/362686.362692

Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. 2015. Variable-order de Bruijn graphs. In *Proceedings of the Data Compression Conference*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer (Eds.). IEEE Computer Society Press, 383–392. DOI: https://doi.org/10.1109/DCC.2015.70

Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. 2012. Succinct de Bruijn graphs. In *WABI 2012: Algorithms in Bioinformatics (Lecture Notes in Computer Science)*, Ben Raphael and Jijun Tang (Eds.), Vol. 7534. Springer-Verlag, 225–235. DOI: https://doi.org/10.1007/978-3-642-33122-0_18

Phelim Bradley, Henk den Bakker, Eduardo Rocha, Gil McVean, and Zamin Iqbal. 2019. Ultrafast search of all deposited bacterial and viral genomic data. *Nat. Biotechnol.* 37 (2019), 152–159. DOI: https://doi.org/10.1038/s41587-018-0010-1

Keith R. Bradnam, Joseph N. Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanç Birol, Sébastien Boisvert, Jarrod A. Chapman, Guillaume Chapuis, Rayan Chikhi et al. 2013. Assemblathon 2: Evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience* 2, 1 (2013). DOI: https://doi.org/10.1186/2047-217X-2-10

Karel Břinda. 2016. *Novel Computational Techniques for Mapping and Classifying Next-generation Sequencing Data*. Ph.D. Dissertation. University of Paris-Est Marne-la-Vallée. DOI: https://doi.org/10.5281/zenodo.1045317

Karel Břinda, Michael Baym, and Gregory Kucherov. 2020. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *bioRxiv* 903443 (2020). DOI: https://doi.org/10.1101/2020.01.12.903443

Andrei Broder and Michael Mitzenmacher. 2003. Network applications of Bloom filters: A survey. *Internet Math.* 1, 4 (2003), 485–509. DOI: https://doi.org/10.1080/15427951.2004.10129096

Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. 2014. On the representation of de Bruijn graphs. In *RECOMB 2014: Research in Computational Molecular Biology (Lecture Notes in Computer Science)*, Roded Sharan (Ed.), Vol. 8394. Springer, 35–55. DOI: https://doi.org/10.1007/978-3-319-05269-4_4

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. DOI: https://doi.org/10.1093/bioinformatics/btw279

Rayan Chikhi and Guillaume Rizk. 2012. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In *WABI 2012: Algorithms in Bioinformatics (Lecture Notes in Computer Science)*, Ben Raphael and Jijun Tang (Eds.), Vol. 7534. Springer, 236–248. DOI: https://doi.org/10.1007/978-3-642-33122-0_19

Justin Chu, Hamid Mohamadi, Emre Erhan, Jeffery Tse, Readman Chiu, Sarah Yeo, and Inanç Birol. 2018. Improving on hash-based probabilistic sequence classification using multiple spaced seeds and multi-index Bloom filters. *bioRxiv* (2018), 434795. DOI: https://doi.org/10.1101/434795

Saar Cohen and Yossi Matias. 2003. Spectral Bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Association for Computing Machinery, 241–252. DOI: https://doi.org/10.1145/872757.872787

Thomas C. Conway and Andrew J. Bromage. 2011. Succinct data structures for assembling large genomes. *Bioinformatics* 27, 4 (2011), 479–486. DOI: https://doi.org/10.1093/bioinformatics/btq697

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* The MIT Press.

Adina Crainiceanu and Daniel Lemire. 2015. Bloofi: Multidimensional Bloom filters. *Inf. Syst.* 54 (2015), 311–324. DOI: https://doi.org/10.1016/j.is.2015.01.002

Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *J. ACM* 21, 2 (1974), 246–260. DOI: https://doi.org/10.1145/321812.321820

Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies.* Association for Computing Machinery, 75–88. DOI: https://doi.org/10.1145/2674005.2674994

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (2000), 281–293. DOI: https://doi.org/10.1109/90.851975

Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Shan Muthukrishnan. 2009. Compressing and indexing labeled trees, with applications. *J. ACM* 57, 1 (2009), 4:1–4:33. DOI: https://doi.org/10.1145/1613676.1613680

Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS'00).* IEEE Computer Society, 390–398. DOI: https://doi.org/10.1109/SFCS.2000.892127

Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2007. Compressed representations of sequences and full-text indexes. *ACM Trans. Algor.* 3, 2 (2007). DOI: https://doi.org/10.1145/1240233.1240243

Hongzhe Guo, Yilei Fu, Yan Gao, Junyi Li, Yadong Wang, and Bo Liu. 2019. deGSM: Memory scalable construction of large scale de Bruijn Graph. *IEEE/ACM Trans. Comput. Biol. Bioinf.* (2019), Early access. DOI: https://doi.org/10.1109/TCBB.2019.2913932

Robert S. Harris and Paul Medvedev. 2020. Improved representation of sequence Bloom trees. *Bioinformatics* 36, 3 (2020), 721–727. DOI: https://doi.org/10.1093/bioinformatics/btz662

Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.* 20, 2 (2002), 192–223. DOI: https://doi.org/10.1145/506309.506312

Guillaume Holley and Páll Melsted. 2019. Bifrost–Highly parallel construction and indexing of colored and compacted de Bruijn graphs. *bioRxiv* (2019), 695338. DOI: https://doi.org/10.1101/695338

Guillaume Holley, Roland Wittler, and Jens Stoye. 2016. Bloom filter trie: An alignment-free and reference-free data structure for pan-genome storage. *Algor. Molec. Biol.* 11, 1 (2016), 3. DOI: https://doi.org/10.1186/s13015-016-0066-8

Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. 2012. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genetics* 44 (2012), 226–232. DOI: https://doi.org/10.1038/ng.1028

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. 2017. ABySS 2.0: Resource-efficient assembly of large genomes using a Bloom filter. *Genome Res.* 27 (2017), 768–777. DOI: https://doi.org/10.1101/gr.214346.116

Daniel Lemire and Owen Kaser. 2010. Recursive *n*-gram hashing is pairwise independent, at best. *Comput. Speech Lang.* 24, 4 (2010), 698–710. DOI: https://doi.org/10.1016/j.csl.2009.12.001

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. DOI: https://doi.org/10.1016/j.ymeth.2016.02.020

Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. 2016. Read mapping on de Bruijn graphs. *BMC Bioinf.* 17 (2016). DOI: https://doi.org/10.1186/s12859-016-1103-9

Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and scalable minimal perfect hashing for massive key sets. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17) (Leibniz International Proceedings in Informatics (LIPIcs))*, Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman (Eds.), Vol. 75. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 25:1–25:16. DOI: https://doi.org/10.4230/LIPIcs.SEA.2017.25

Xinan Liu, Ye Yu, Jinpeng Liu, Corrine F. Elliott, Chen Qian, and Jinze Liu. 2018. A novel data structure to support ultra-fast taxonomic classification of metagenomic sequences with k-mer signatures. *Bioinformatics* 34, 1 (2018), 171–178. DOI: https://doi.org/10.1093/bioinformatics/btx432

Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-scale Algorithm Design*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9781139940023

Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. 2005. An extension of the Burrows Wheeler transform to k words. In *Proceedings of the Data Compression Conference*, James A. Storer and Martin Cohn (Eds.). IEEE Computer Society Press, 469. DOI: https://doi.org/10.1109/DCC.2005.13

Camille Marchet, Christina Boucher, Simon J. Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. 2019a. Data structures based on k-mers for querying large collections of sequencing datasets. *bioRxiv* 866756 (2019). DOI: https://doi.org/10.1101/866756

Camille Marchet, Maël Kerbiriou, and Antoine Limasset. 2019b. Indexing de Bruijn graphs with minimizers. *bioRxiv* (2019), 546309. DOI: https://doi.org/10.1101/546309

Camille Marchet, Lolita Lecompte, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo. 2018. A resource-frugal probabilistic dictionary and applications in bioinformatics. *Disc. Appl. Math.* 274 (2018), 92–102. DOI: https://doi.org/10.1016/j.dam.2018.03.035

Paul Medvedev, Rayan Chikhi, and Antoine Limasset. 2019. Bi-directed graphs in BCALM 2. Retrieved from https://github.com/GATB/bcalm/blob/master/bidirected-graphs-in-bcalm2/bidirected-graphs-in-bcalm2.md.

Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. 2007. Computability of models for sequence assembly. In *WABI 2007: Algorithms in Bioinformatics (Lecture Notes in Computer Science)*, Raffaele Giancarlo and Sridhar Hannenhalli (Eds.), Vol. 4645. Springer, 289–301. DOI: https://doi.org/10.1007/978-3-540-74126-8_27

Michael Mitzenmacher. 2002. Compressed Bloom filters. *IEEE/ACM Trans. Netw.* 10, 5 (2002), 604–612. DOI: https://doi.org/10.1109/TNET.2002.803864

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. DOI: https://doi.org/10.1093/bioinformatics/btx067

Harun Mustafa, Ingo Schilken, Mikhail Karasikov, Carsten Eickhoff, Gunnar Rätsch, and André Kahles. 2019. Dynamic compression schemes for graph coloring. *Bioinformatics* 35, 3 (2019), 407–414. DOI: https://doi.org/10.1093/bioinformatics/bty632

Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9781316588284

Gonzalo Navarro and Kunihiko Sadakane. 2014. Fully functional static and dynamic succinct trees. *ACM Trans. Algor.* 10, 3 (2014), 16:1–16:39. DOI: https://doi.org/10.1145/2601073

Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. Society for Industrial and Applied Mathematics, 60–70. DOI: https://doi.org/10.5555/2791188.2791194

Tony Pan, Rahul Nihalani, and Srinivas Aluru. 2020. Fast de Bruijn graph compaction in distributed memory environments. *IEEE/ACM Trans. Comput. Biol. Bioinf.* 17, 1 (2020), 136–148. DOI: https://doi.org/10.1109/TCBB.2018.2858797

Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2018. Mantis: A fast, small, and exact large-scale sequence search index. *Cell Syst.* (2018), 201–207. DOI: https://doi.org/10.1016/j.cels.2018.05.021

Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017c. deBGR: An efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, 14 (2017), i133–i141. DOI: https://doi.org/10.1093/bioinformatics/btx261

Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017a. A general-purpose counting filter: Making every bit count. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. Association for Computing Machinery, 775–787. DOI: https://doi.org/10.1145/3035918.3035963

Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017b. Squeakr: An exact and approximate k-mer counting system. *Bioinformatics* 34, 4 (2017), 568–575. DOI: https://doi.org/10.1093/bioinformatics/btx636

David Pellow, Darya Filippova, and Carl Kingsford. 2017. Improving Bloom filter performance on sequence data using k-mer Bloom filters. *J. Comput. Biol.* 24, 6 (2017), 547–557. DOI: https://doi.org/10.1089/cmb.2016.0155

Amatur Rahman and Paul Medvedev. 2020. Representation of k-mer sets using spectrum-preserving string sets. *bioRxiv* 896928 (2020). DOI: https://doi.org/10.1101/2020.01.07.896928

Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algor.* 3, 4 (2007), 43. DOI: https://doi.org/10.1145/1290672.1290680

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

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

Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Association for Computing Machinery, 76–85. DOI: https://doi.org/10.1145/872757.872770

Alexander Sczyrba, Peter Hofmann, Peter Belmann, David Koslicki, Stefan Janssen, Johannes Dröge, Ivan Gregor, Stephan Majda, Jessika Fiedler, Eik Dahms et al. 2017. Critical assessment of metagenome interpretation—A benchmark of metagenomics software. *Nat. Methods* 14, 11 (2017), 1063–1071. DOI: https://doi.org/10.1038/nmeth.4458

Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Muller-Wittig. 2010. A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. *J. Comput. Biol.* 17, 4 (2010), 603–615. DOI: https://doi.org/10.1089/cmb.2009.0062

Jared T. Simpson and Mihai Pop. 2015. The theory and practice of genome sequence assembly. *Ann.l Rev. Genom. Hum. Genet.* 16 (2015), 153–172. DOI: https://doi.org/10.1146/annurev-genom-090314-050032

Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.* 34, 3 (2016), 300–302. DOI: https://doi.org/10.1038/nbt.3442

Brad Solomon and Carl Kingsford. 2018. Improved search of large transcriptomic sequencing databases using split sequence Bloom trees. *J. Comput. Biol.* 25, 7 (2018), 755–765. DOI: https://doi.org/10.1089/cmb.2017.0265

Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. 2010. Classification of DNA sequences using Bloom filters. *Bioinformatics* 26, 13 (2010), 1595–1600. DOI: https://doi.org/10.1093/bioinformatics/btq230

Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. 2018. AllSome Sequence Bloom Trees. *J. Comput. Biol.* 25, 5 (2018), 467–479. DOI: https://doi.org/10.1089/cmb.2017.0258

Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of Bloom filters for distributed systems. *IEEE Commun. Surv. Tutor.* 14, 1 (2011), 131–155. DOI: https://doi.org/10.1109/SURV.2011.031611.00024

Niko Välimäki and Eric Rivals. 2013. Scalable and versatile k-mer indexing for high-throughput sequencing data. In *Proceedings of the 9th International Symposium on Bioinformatics Research and Applications (ISBRA'13) (Lecture Notes in Computer Science)*, Vol. 7875. Springer, 237–248. DOI: https://doi.org/10.1007/978-3-642-38036-5_24

Derrick E. Wood and Steven L. Salzberg. 2014. Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biol.* 15 (2014), R46. DOI: https://doi.org/10.1186/gb-2014-15-3-r46

Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Chen Qian, and Jinze Liu. 2018. SeqOthello: Query over RNA-seq experiments at scale. *Genome Biol.* 19 (2018). DOI: https://doi.org/10.1186/s13059-018-1535-9

Jens Zentgraf, Henning Timm, and Sven Rahmann. 2020. Cost-optimal assignment of elements in genome-scale multi-way bucketed Cuckoo hash tables. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, Guy Blelloch and Irene Finocchi (Eds.). SIAM, 186–198. DOI: https://doi.org/10.1137/1.9781611976007.15

Received April 2019; revised June 2020; accepted October 2020