JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 27, Number 4, 2020

Pp. 485–499
DOI: 10.1089/cmb.2019.0322

An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search

FATEMEH ALMODARESI,[1] PRASHANT PANDEY,[2] MICHAEL FERDMAN,[3] ROB JOHNSON,[3,4] and ROB PATRO[1]

ABSTRACT

The colored de Bruijn graph (cdbg) and its variants have become an important combinatorial structure used in numerous areas in genomics, such as population-level variation detection in metagenomic samples, large-scale sequence search, and cdbg-based reference sequence indices. As samples or genomes are added to the cdbg, the color information comes to dominate the space required to represent this data structure. In this article, we show how to represent the color information efficiently by adopting a hierarchical encoding that exploits correlations among color classes—patterns of color occurrence—present in the de Bruijn graph (dbg). A major challenge in deriving an efficient encoding of the color information that takes advantage of such correlations is determining which color classes are close to each other in the high-dimensional space of possible color patterns. We demonstrate that the dbg itself can be used as an efficient mechanism to search for approximate nearest neighbors in this space. While our approach reduces the encoding size of the color information even for relatively small cdbgs (hundreds of experiments), the gains are particularly consequential as the number of potential colors (i.e., samples or references) grows into thousands. We apply this encoding in the context of two different applications; the implicit cdbg used for a large-scale sequence search index, Mantis, as well as the encoding of color information used in population-level variation detection by tools such as Vari and Rainbowfish. Our results show significant improvements in the overall size and scalability of representation of the color information. In our experiment on 10,000 samples, we achieved >11 × better compression compared to Ramen, Ramen, Rao (RRR).

Keywords: compression schemes, de bruijn graph, proximate membership query, RNA-sequence search.

[1]Department of Computer Science, University of Maryland, College Park, Maryland.
[2]School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.
[3]Department of Computer Science, Stony Brook University, Stony Brook, New York.
[4]VMware Research, Palo Alto, California.

1. INTRODUCTION

The colored de Bruijn graph (cdbg) (Iqbal Zamin et al., 2012), an extension of the classical de Bruijn graph (dbg) (Pevzner and Tang, 2001; Pevzner et al., 2001; Chikhi et al., 2014), is a key component of a growing number of genomics tools. Augmenting the traditional dbg with ''color'' information provides a mechanism to associate meta-data, such as the raw sample or reference of origin, with each *k*-mer. Coloring the dbg enables it to be used in a wide range of applications, such as large-scale sequence search (Solomon and Kingsford, 2016, 2017; Bradley et al., 2017; Sun et al., 2017; Pandey et al., 2018) [although some (Solomon and Kingsford, 2016, 2017; Sun et al., 2017) do not explicitly couch their representations in the language of the cdbg], population-level variation detection (Holley et al., 2016; Almodaresi et al., 2017; Muggli et al., 2017), traversal and search in a pan-genome (Holley et al., 2016), and sequence alignment (Liu et al., 2016a). The popularity and applicability of the cdbg have spurred research into developing space-efficient and high-performance data-structure implementations.

An efficient and fast representation of cdbg requires optimizing both the dbg and the color information. While there exist efficient and scalable methods for representing the topology of the dbg (Bowe et al., 2012; Chikhi and Rizk, 2012; Chikhi et al., 2014; Salikhov et al., 2014; Crawford et al., 2018; Pandey et al., 2017a) with fast query time, a scalable and exact representation of the color information has remained a challenge. Recently, Mustafa et al. (2019) have tackled this challenge by relaxing the exactness constraints—allowing the returned color set for a *k*-mer to contain extra samples with some controlled probability—but it is not immediately clear how this method can be made exact.

Specifically, existing exact color representations suffer from large sizes and a fast growth rate that leads them to dominate the total representation size of the cdbg with even a moderate number of input samples (Fig. 3b). As a result, the color information grows to dominate the space used by all these indexes and limits their ability to scale to large input data sets.

Iqbal et al. (2012) introduced cdbgs and proposed a hash-based representation of the dbg in which each *k*-mer is additionally tagged with the list of reference genomes in which it is contained. Muggli et al. (2017) reduced the size of the cdbg in VARI by replacing the hash map with BOSS (Bowe et al. 2012) [a BWT-based (Burrows and Wheeler, 1994) encoding of the dbg that assigns a unique ID to each *k*-mer] and using a Boolean matrix indexed by the unique *k*-mer ID and genome reference ID to indicate occurrence. They reduced the size of the occurrence matrix by applying off-the-shelf compression techniques Ramen, Ramen, Rao (RRR) (Raman et al., 2002) and Elias-Fano (Elias, 1974) encoding. Rainbowfish Almodaresi et al. (2017) shrank the color table further by ensuring that rows of the color matrix are unique, mapping all *k*-mers with the same color information to a single row and assigning row indices based on the frequency of each occurrence pattern. However, despite these improvements, the scalability of the resulting structure remains limited because even after eliminating redundant colors, the space for the color table grows quickly to dominate the total space used by these data structures.

We observe that, in real biological data, even when the number of distinct color classes is large, many of them will be near each other in terms of the set of samples or references they encode. That is, the color classes tend to be highly correlated rather than uniformly spread across the space of possible colors. There are intuitive reasons for such characteristics. For example, we observe that adjacent *k*-mers in the dbg are extremely likely to have either identical or similar color classes, enabling storage of small deltas instead of the complete color classes. This is because *k*-mers adjacent in the dbg are likely to be adjacent (and hence present) in a similar set of input samples. In the context of sequence search, because genomes and transcriptomes are largely preserved across organs, individuals, and even across related species, we expect two *k*-mers that occur together in one sample to be highly correlated in their occurrence across many samples. Thus, we can take advantage of this correlation when devising an efficient encoding scheme for the cdbg's associated color information.

In this article, we develop a general scheme for efficient and scalable encoding of the color information in the cdbg by encoding color classes (i.e., the patterns of occurrence of a *k*-mer in samples) in terms of their differences (which are small) with respect to some ''neighboring'' color class. The key technical challenge, solved by our work, is efficiently searching for the neighbors of color classes in the high-dimensional space of colors by leveraging the observation that similar color classes tend to be topologically close in the underlying dbg. We construct a weighted graph on the color classes in the cdbg, where the weight of each edge corresponds to the space required to store the delta between its end points. Finding the

minimum spanning tree (MST) of this graph gives a minimal delta-based representation. Although reconstructing a color class on this representation requires a walk to the MST root node, abundant temporal locality on the lookups allows us to use a small cache to mitigate the performance impact, yielding query throughput that is essentially the same as when all color classes are represented explicitly.

An alternative would have been to try to limit the depth (or diameter) of the MST. This problem is heavily studied in two forms: the unrooted bounded-diameter MST problem (Raidl, 2008) and the rooted hop-constrained MST problem (Althaus et al., 2005). Neither is in APX, that is, it is not possible to approximate them to within any constant factor (Manyem and Stallmann, 1996). Althaus et al. (2005) gave an $O(\log n)$ approximation assuming the edge weights form a metric. Khuller et al. (2002) show that, if the edge *lengths* are the same as the edge *weights*, then there is an efficient algorithm for finding a spanning tree that is within a constant of optimal in terms of both diameter and weight. Marathe et al. (1998) show that in general we can find trees within $O(\log n)$ of the minimum diameter and weight. We can't use Khuller's approach (because our edge lengths are not equal to our edge weights), and even a $O(\log n)$ approximation would give up a potentially substantial amount of space.

We showcase the generality and applicability of our color class table compression technique by demonstrating it in two computational biology applications: sequence search and variation detection. We compare our novel color class table representation with the representation used in Mantis (Pandey et al., 2018), a state-of-the-art large-scale sequence-search tool that uses a cdbg to index a set of sequencing samples, and the representation used in Rainbowfish (Almodaresi et al., 2017), a state-of-the-art index to facilitate variation detection over a set of genomes. We show that our approach maintains the same query performance while achieving over $11\times$ and $2.5\times$ storage savings relative to the representation previously used by these tools.

2. METHODS

This section describes our compact cdbg representation. We first define cdbgs and briefly describe existing compact cdbg representations. We then outline the high-level idea behind our compact representation and explain how we use the dbg to efficiently build our compact representation. Finally, we describe implementation details and optimizations to our query algorithm.

2.1. Colored de Bruijn graphs

The dbgs are widely used to represent the topological structure of a set of k-mers (Pevzner et al., 2001; Zerbino and Birney, 2008; Simpson et al., 2009; Grabherr et al., 2011; Schulz et al., 2012; Chang et al., 2015; Liu et al., 2016b; Pandey et al., 2017a). The dbg induced by a set of k-mers is defined below.

Definition 1. *Given a set E of k-mers, the dbg induced by E has edge set E, where each k-mer (or edge) connects its two $(k-1)$-length substrings (or vertices).*

Cdbgs extend the dbg by assigning a *color class* $C(x)$ to each edge (or node) x of the dbg. The color class $C(x)$ is a set drawn from some universe U. Examples of U and $C(x)$ are

- Sometimes, U is a set of reference genomes, and $C(x)$ is the subset of reference genomes containing k-mer x (Liu et al., 2016a; Almodaresi et al., 2017, 2018; Muggli et al., 2017).
- Sometimes, U is a set of *reads*, and $C(x)$ is the subset of reads containing x (Alipanahi et al., 2018a,b; Turner et al., 2018).
- Sometimes, U is a set of sequencing experiments, and $C(x)$ is the subset of sequencing experiments containing x (Solomon and Kingsford, 2016, 2017; Sun et al., 2017; Pandey et al., 2018).

The goal of a cdbg representation is to store E and C as compactly as possible[1], while supporting the following operations efficiently:

- *Point query.* Given a k-mer x, determine whether x is in E.
- *Color query.* Given a k-mer $x \in E$, return $C(x)$.

[1]The nodes of the de Bruijn graph are typically stored implicitly, because the node set is simply a function of E.

Given that we can perform point queries, we can traverse the dbg by simply querying for the eight possible predecessor/successor edges of an edge. This enables us to implement more advanced algorithms, such as bubble calling (Iqbal Zamin et al., 2012).

Many cdbg representations typically decompose, at least logically, into two structures: one structure storing a dbg and associating an ID with each k-mer, and one structure mapping these IDs to the actual color class (Almodaresi et al., 2017; Muggli et al., 2017; Pandey et al., 2017b). The individual color classes can be represented as bit vectors, lists, or using a hybrid scheme (Yu et al., 2018). This information is typically compressed (Ziv and Lempel, 1977; Raman et al., 2002; Ottaviano and Venturini, 2014).

Our article follows this standard approach and focuses exclusively on reducing the space required for the structure storing the color information. We propose a compact representation that, given a color ID, can return the corresponding color efficiently. Although we pair our color table representation with the dbg structure representation of the counting quotient filter (CQF) (Pandey et al., 2017b) as used in Mantis (Pandey et al. 2018), our proposed color table representation can be paired with other dbg representations.

2.2. A similarity-based cdbg representation

The key observation behind our compressed color-class representation is that the color classes of k-mers that are adjacent in the dbg are likely to be very similar. Thus, rather than storing each color class explicitly, we can store only a few color classes explicitly and, for all the remaining color classes, we store only their differences from other color classes. Because the differences are small, the total space used by the representation will be small.

Motivated by the observation above, we propose to find an encoding of the color classes that takes advantage of the fact that most color classes can be represented in terms of only a small number of edits (i.e., flipping the parity of only a few bits) with respect to some neighbor in the high-dimensional space of the color classes. This idea was first explored by Bookstein and Klein (1991) in the context of information retrieval. Bookstein and Klein showed how to exploit the implicit clustering among bitmaps in Information Retrieval (IR) to achieve excellent reduction in storage space to represent those bitmaps using an MST as the underlying representation. Unfortunately, the approach taken by Bookstein and Klein cannot be directly used in our problem, since it requires computing and optimizing upon the full Hamming distance graph of the bit vectors being represented, which is not tractable for the scale of data we are analyzing. Hence, what we need is a method to efficiently discover an incomplete and highly-sparse Hamming distance graph that, nonetheless, supports a low-weight spanning tree. We describe below how we apply and modify this approach in the context of the set of correlated bit vectors (i.e., color classes) that we wish to encode.

We construct our compressed color class representation as follows (Fig. 1). For each edge x of the dbg, let $C(x)$ be the color class of x. Let $\mathcal{C}$ be the set of all color classes that occur in the dbg. We first construct an undirected graph with vertex set $\mathcal{C}$ and edge set reflecting the adjacency relationship implied by the dbg. In other words, there is an edge between color classes c_1 and c_2 if there exist adjacent edges (i.e., incident on the same node) x and y in the dbg such that $c_1 = C(x)$ and $c_2 = C(y)$. These edges indicate color classes that are likely to be similar, based on the structure of the dbg. We then add a special node $\emptyset$ to the color class graph, which is connected to every node. We set the weight of every edge in the color class graph to be the Hamming distance between its two end points (where we view color classes as bit vectors and $\emptyset$ is the all-zeros bit vector).

We then compute a MST of the color class graph and root the tree at the special $\emptyset$ node. Note that, because the $\emptyset$ node is connected to every other node in the graph, the graph is connected and hence an MST is guaranteed to exist. Using a MST, we minimize the total size of the differences that we need to store in our compressed representation.

We then store the MST as a table mapping each color class ID to the ID of its parent in the MST, along with a list of the differences between the color class and its parent. For convenience we can view the list of differences between color class c_1 and color class c_2 as a bit vector $c_1 \oplus c_2$, where $\oplus$ is the bit-wise exclusive-or operation. To reconstruct a color class given its ID i, we simply xor all the difference vectors we encounter while walking from i to the root of the MST.

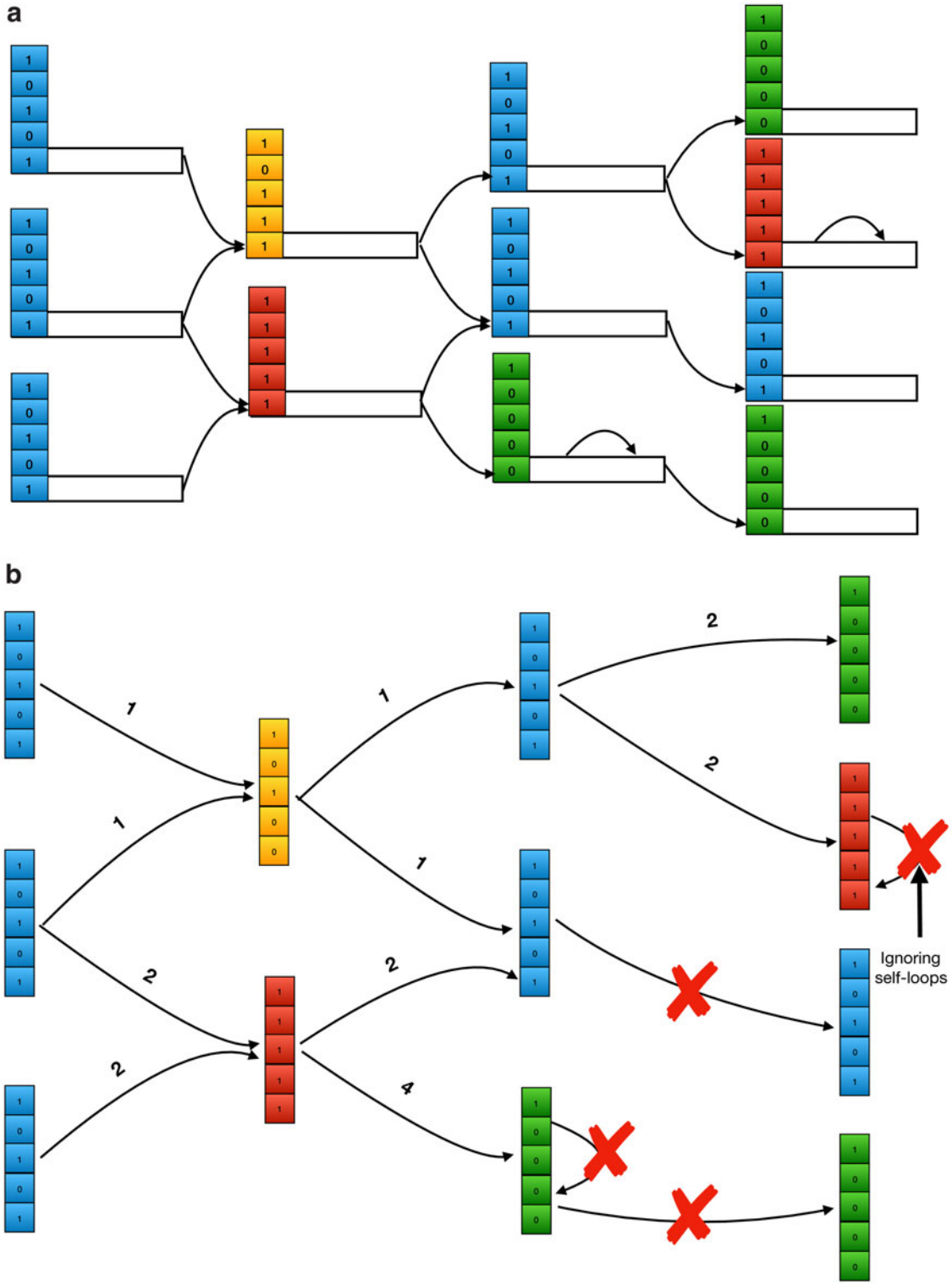

FIG. 1. Encoding color classes by finding the MST of the color class graph, an undirected graph derived from cdbg. The order of the process is **(a–c)**. The arrows in **(a, b)** show the direction of edges in the dbg which is a directed graph. The optimal achievable MST is shown in **(d)** for comparison. Since we never observe the edge between any *k*-mers from color classes green and yellow in cdbg, we won't have the edge between color classes green and yellow, and therefore, our final MST is not equal to the best MST we can get from a complete color class graph. cdbg, colored de Bruijn graph; MST, minimum spanning tree.

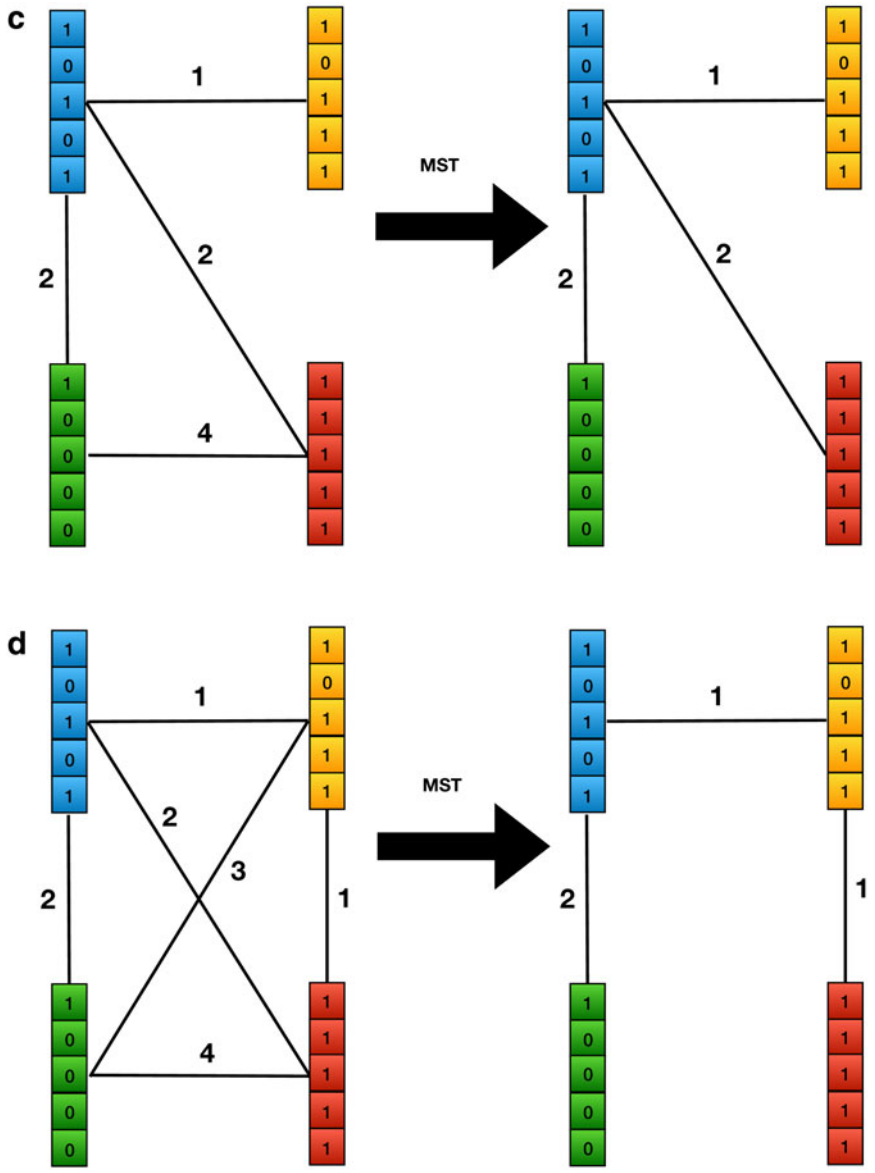

FIG. 1. (*Continued*).

2.3. Implementation of the MST data structure

Assuming we have $|\mathcal{C}|$ color classes, $|U|$ colors, and an MST with total weight of w over the color class graph, we store all the information required to retrieve the original color bit vector for each color class ID based on the MST structure into three data structures:

- Parent vector: This vector contains $|\mathcal{C}|$ slots, each of size $\lceil \log_2 \mathcal{C} \rceil$. The value stored in index i represents the parent color class ID of the color class with index i in the MST.
- Delta vector: This vector contains w slots, each of size $\lceil \log_2 |U| \rceil$. For each pair of parent and child in the parent vector, we compute a vector of the indices at which they differ. The delta vector is the concatenation of these per-edge delta vectors, ordered by the ID of the source of the edge. Note that the per-edge delta vectors will not all be of the same length, because some edges have larger weight than others. Thus, we need an auxiliary data structure to record the boundaries between the per-edge deltas within the overall delta vector.

- Boundary bit vector: This vector contains w bits, where a set bit indicates the boundary between two delta sets within the delta vector. To find the starting position, within the delta vector, of the per-edge delta list for the MST edge with source ID i, we perform *select(i)* on the boundary vector. Select returns the position of the ith one in the boundary vector.

2.3.1. Query of the MST-based representation. Figure 2 shows how queries proceed using this encoding. We start with an empty accumulator bit vector and a color class ID i for which we want to compute the corresponding color class. We perform a select query for i and $i+1$ in the boundary bit vector to get the boundaries of i's difference list in the delta vector. We then iterate over its difference list and flip the indicated bits in our accumulator. We then set $i \leftarrow \text{PARENT}[i]$ and repeat until i becomes 0, which indicates that we have reached the root. At this point, the accumulator will be equal to the bit vector for color class i.

2.4. Integration in Mantis

Once constructed, our MST-based color class representation is a drop-in replacement for the current color class representations used in several existing tools, including Mantis (Pandey et al. 2018) and Rainbowfish (Almodaresi et al. 2017). Their existing color class tables support a single operation—querying for a color class by its ID—and our MST-based representation supports exactly the same operation.

For this article, we integrated our MST-based representation into Mantis. The same space savings can be achieved in other tools, particularly Rainbowfish, which has a similar color-class encoding as Mantis.

2.4.1. Construction. We construct our MST-based color-class representation as follows. First, we run Mantis to build its default representation of the cdbg. We then build the color-class graph by walking the dbg and adding all the corresponding edges to the color-class graph. The edge set is typically much smaller

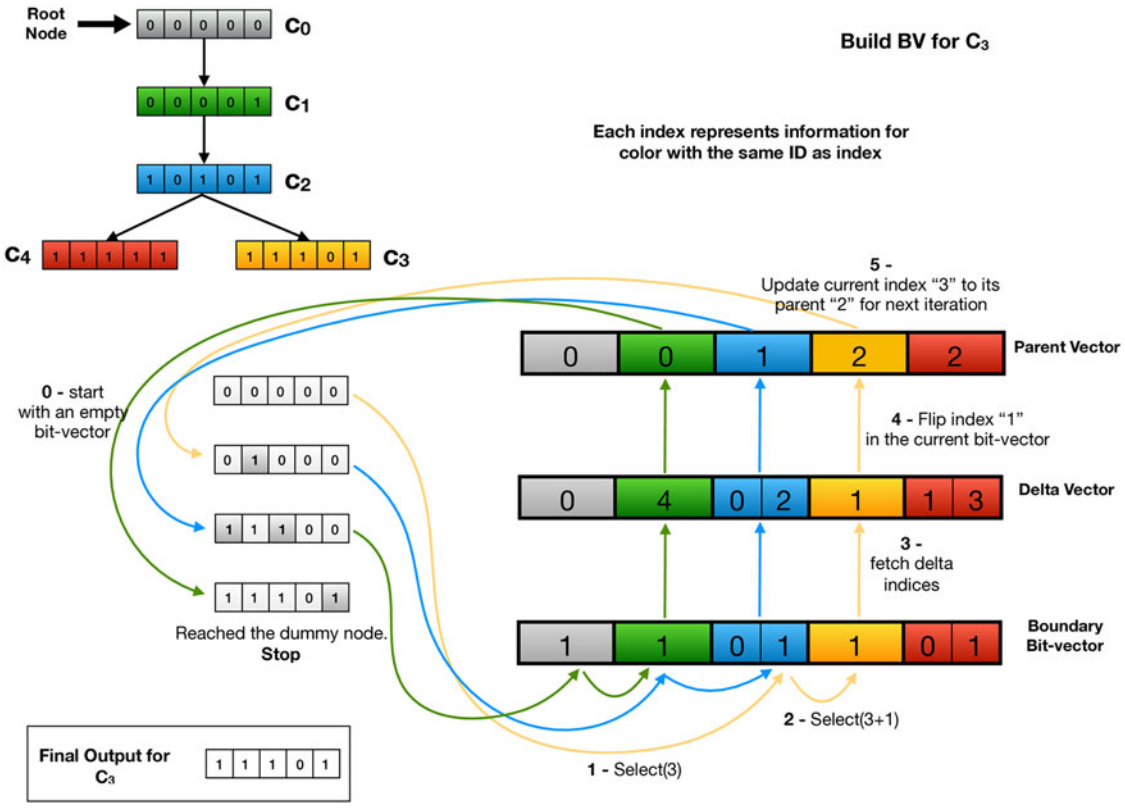

FIG. 2. The conceptual MST (top-left); the data structure to store the color information in the format of an MST (right). This figure also illustrates the steps required to build one of the color vectors (C_3) at the leaf of the tree. Note that the query process shown here does not depict the caching policy we apply in practice.

than the dbg (because many dbg edges may map to the same edge in the color-class graph), so this can be done in RAM. Note that we do not compute the weights of the edges during this pass, because that would require having all of the large color-class bit vectors in memory to compute their Hamming distance.

In the second pass, we traverse the edge set and compute the weight of each edge. To minimize RAM usage during this phase, we sort the edges and iterate over them in a ''blocked'' manner. Specifically, Mantis stores the color class bit vectors on-disk sequentially by ID, grouped into blocks of roughly 6 GBs each. We sort the edges lexicographically by their source and destination block. We then load all pairs of blocks and compute the weights of all the edges between the two blocks currently in memory. At all times, we need only two blocks of color class vectors in memory. Given the weighted graph, we compute the MST and make one final pass to determine the relevant delta lists and encode our final MST structure.

2.4.2. Parallelization. We note that, after having constructed the Mantis representation, most phases of the MST construction algorithm are trivially parallelized. MST construction decomposes into three phases: (1) color-class graph construction, (2) MST computation, and (3) color-class representation generation. We parallelize graph construction and color-class representation generation. The MST computation itself is not parallelized.

We parallelized the determination of edges in the color-class graph by assigning each thread a range of the k-mer-to-color-class-ID map. Each thread explores the neighbors of the k-mers that appear in its assigned range, and any redundant edges are deduplicated when all threads are finished. Similarly, we parallelized the computation of edge weights and the extraction of the delta vectors that correspond to each edge in the MST. Given the list of edges sorted lexicographically by their end points (determined during the first phase), it is straightforward to partition the work for processing batches of edges across many threads. It is possible, of course, that the batches will display different workloads and that some threads will complete their assigned work before others. We have not yet made any attempt to optimize the parallel construction of the MST in this regard, although many such optimizations are likely possible.

2.4.3. Accelerating queries with caching. The encoded MST is not a balanced tree, so decoding a color bit vector might require walking a long path to the root, which negatively impacts the query time. Attempting to explicitly minimize the depth or diameter of the MST is, as discussed in Section 1, not generally approximable within a constant factor. However, considering the fact that the frequency distribution of the color classes is very skewed, some of the color classes are more popular or have more children and, therefore, are in the path of many more nodes. We take advantage of these data characteristics by caching the most recent queried color bit vectors. Every time we walk up the tree, if the color bit vector for a node is already in the cache, our query algorithm stops at that point and applies all the deltas to this bit vector instead of the zero bit vector of the root. This caching approach significantly improves the query time, resulting in the final query time required to decode a color class being marginally faster than direct RRR access.

The cache policy is designed with the tree structure of our color-class representation in mind. Specifically, we want to cache nodes near the leaves, but not so close to the leaves that we end up caching essentially the entire tree. In addition, we don't want to cache infrequently queried nodes. Thus we use the following caching policy: all queried nodes are cached. Furthermore, we cache interior nodes visited during a query as follows. If a query visits a node that has been visited by >10 other queries and is >10 hops away from the currently queried item, then we add that node to the cache. If a query visits more than one such node, we add the first one encountered.

In our experiments, we used a cache of 10,000 nodes and managed the cache using a FIFO policy.

2.5. Comparison with brute-force and approximate-nearest-neighbor-based approaches

Our MST-based color-class representation uses the dbg as a hint as to which color classes are likely to be similar. This leads to the natural question: how good are the hints provided by the dbg?

One could imagine alternatively constructing the MST on the complete color-class graph. This would yield the absolutely lowest-weight spanning tree on the color classes. Unfortunately, no MST algorithm runs in less than $\Omega(|E|)$ time, so this would make our construction time quadratic in the number of color classes. The number of color classes in our experiments ranges from 10^6 to 10^9, so the number of edges in the complete color-class graph would be in the order of 10^{12} to 10^{18}, or possibly even more, making this algorithm impractical for the largest data sets considered in this article.

Alternatively, we could try to use an approximate nearest-neighbor algorithm to find pairs of color classes with small Hamming distance. As an experiment, we implemented an approximate nearest neighbor algorithm that bucketed color classes by their projection into a smaller-dimensional subspace. Nearest-neighbor queries were computed by searching within the queried item's bucket. Results were disappointing. Even on small data sets, the average distance between the queried item and the returned neighbor was several times larger than the average distance found using the neighbors suggested by the dbg. Thus, we did not pursue this direction further.

3. EVALUATION

In this section we evaluate our MST-based representation of the color information in the cdbg. All our experiments use Mantis with our integrated MST-based color-class representation.

Evaluation metrics: We evaluate our MST-based representation on the following parameters:

- Scalability. How does our MST-based color-class representation scale in terms of space with increasing number of input samples, and how does it compare to the existing representations of Mantis?
- Construction time. How long does it take—in addition to the original construction time for building cdbg—to build our MST-based color-class representation?
- Query performance. How long does it take to query the cdbg using our MST-based color-class representation?

3.1. Experimental procedure

3.1.1. System specifications. Mantis takes as input a collection of *squeakr* files (Pandey et al., 2017c). Squeakr is a *k*-mer counter that takes as input a collection of fastq files and produces as output a single file with a compact hash table mapping each *k*-mer to the number of times it occurs in the input files. As is standard in evaluations of large-scale sequence search indexes, we do not benchmark the time required to construct these filters.

The data input to the construction process was stored on four-disk mirrors (eight disks total). Each is a Seagate 7200 rpm 8 TB disk (ST8000VN0022). They were formatted using ZFS and exported using NFS over a 10 Gb link. We used different systems to run and evaluate time, memory, and disk requirements for the two steps of preprocessing and index building as were done by Pandey et al. (2018).

For index building and query benchmarks, we ran all the experiments on the same system used in Mantis (Pandey et al., 2018), an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20 GHz with 44 cores and 56 MB L3 cache) with 512 GB RAM and a 4 TB TOSHIBA MG03ACA4 ATA HDD running Ubuntu 16.10 (Linux kernel 4.8.0-59-generic). Constructing the main index was done using a single thread, and the MST construction was performed using 16 threads. Query benchmarks were also performed using a single thread.

3.1.2. Data to evaluate scalability and comparison to Mantis. We integrated and evaluated our MST-based color-class representation within Mantis, so we briefly review Mantis here. Mantis builds an index on a collection of unassembled raw sequencing data sets. Each data set is called a *sample*. The Mantis index enables fast queries of the form, ''Which samples contain this *k*-mer,'' and ''Which samples are likely to contain this string of bases?'' Mantis takes as input one *squeakr* file per sample (Pandey et al., 2017c). A squeakr file is a compact hash table mapping each *k*-mer to the number of times it occurs within that sample. Squeakr also has the ability to serialize a hash that simply represents the set of *k*-mers present at or above some user-provided threshold; we refer to these as filtered Squeakr files. Using the filtered Squeakr files vastly reduces the required intermediate storage space and also decreases the construction time required for Mantis considerably. For example, for the breast, blood, and brain data set (2586 samples), the unfiltered Squeakr files required: 2.5 TB of space, while the filtered files require only: 108 GB. To save intermediate storage space and speed index construction, we built our Mantis representation from these filtered Squeakr files.

Given the input files, Mantis constructs an index consisting of two files: a map from *k*-mer to color-class ID and a map from color-class ID to the bit vector encoding that color class. The first map is stored as a CQF, which is the same compact hash table used by Squeakr. The color-class map is an RRR-compressed bit vector.

Recall that our construction process is implemented as a postprocessing step on the standard Mantis color-class representation. For construction times, we report only this postprocessing step. This is because our MST-based color-class representation is a generic tool that can be applied to many cdbg representations other than Mantis, so we want to isolate the time spent on MST construction.

To test the scalability of our new color-class representation, we used a randomly selected set of 10,000 paired-end, human, bulk RNA-seq short-read experiments downloaded from European Nucleotide Archive (NIH, 2017) in gzipped FASTQ format. In addition, we have built the proposed index for 2586 sequencing samples from human blood, brain, and breast (BBB) tissues originally used by Solomon and Kingsford (2016) and also used in the subsequent work (Solomon and Kingsford, 2017; Sun et al., 2017; Yu et al., 2018), including Mantis (Pandey et al., 2018), as a point of comparison with these representations. The set of 10,000 experiments does not overlap with the BBB samples. The full list of 10,000 experimental identifiers can be obtained from https://github.com/COMBINE-lab/color-mst/blob/master/input_lists/nobbb10k_shuffled.lst The total size of all these experiments (gzipped) is 25.23 TB.

To eliminate spurious k-mers that occur with insignificant abundance within a sample, the squeakr files are filtered to remove low-abundance k-mers. We adopted the same cutoff policy originally proposed by Solomon and Kingsford (2016), by discarding k-mers that occur less than some threshold number of times. The thresholds are determined according to the size (in bytes) of the gzipped sample, and the thresholds are given in Table 1. We adopt a value of $k = 23$ for all experiments.

3.2. Evaluation results

3.2.1. Scalability of the new color-class representation. Figure 3a and Table 2 show how the size of our MST-based color-class representation scales as we increase the number of samples indexed by Mantis. For comparison, we also give the size of Mantis' RRR-compression-based color-class representation. Figure 3a also plots the size of the CQF that Mantis uses to map k-mers to color class IDs. We can draw several conclusions from these data:

- The MST-based representation is an order-of-magnitude smaller than the RRR-based representation.
- The gap between the RRR-based representation and the MST-based representation grows as we increase the number of input samples. This suggests that the MST-based representation grows asymptotically slower than the RRR-based representation.
- The MST-based color-class representation is, for large numbers of samples, about $5\times$ smaller than the CQF. This means that representing the color classes is no longer the scaling bottleneck.

Table 2 also shows the scaling rate of all elements of the MST representation, in addition to the ratio of MST over the color bit vector. As expected, the list of deltas dominates the MST representation both in terms of total size and in terms of growth. Table 2 also shows the average edge weight of the edges in the MST. The edge weight grows approximately proportional to $\Theta(\log(\#\text{ of samples}))$ (i.e., every time we double the number of samples, the average edge weight increases by almost exactly 1). This suggests that our dbg-based algorithm is able to find pairs of similar color classes. The time column shows the time required to build the MST representation (which is in addition to the Mantis construction time required to produce the input to the MST compression algorithm).

To better understand the scaling of the different components of a cdbg representation, we plot the sizes of the RRR-based color-class representations and MST-based representations on a log–log scale in Figure 3b. Based on the data, the RRR-based representation appears to grow in size at a rate of roughly $\Theta(n^{1.5})$,

Table 1. Minimum Number of Times a k-Mer Must Appear in an Experiment to Be Counted as Abundantly Represented in That Experiment (Taken from the Sequence Bloom Tree Article)

Min size	*Max size*	*Cutoff*	*No. of experiments with specified threshold*
0	≤300 MB	1	2784
>300 MB	≤500 MB	3	798
>500 MB	≤1 GB	10	1258
>1 GB	≤3 GB	20	2296
>3 GB	∞	50	2864

Note, the k-mers with count of "cutoff" *are* included at each threshold.

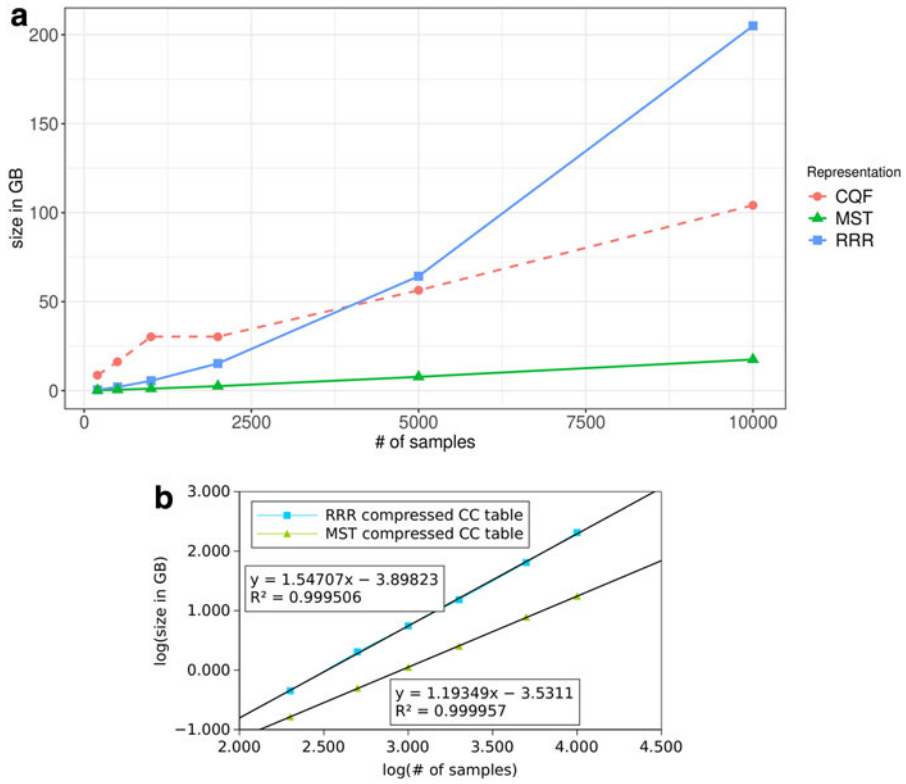

FIG. 3. Size of the MST-based color-class representation versus the RRR-based color-class representation. **(a)** Sizes of the RRR and MST-based color class representations with respect to the number of samples indexed from the human bulk RNA-seq data set. The counting quotient filter component is the Mantis representation of the de Bruijn graph. **(b)** Empirical asymptotic analysis of the growth rates of the sizes of RRR-based color class representation and the MST-based color class representation. The RRR-based representation grows at a rate of $\approx\Theta(n^{1.5})$, where n is the number of samples. The MST-based representation grows at a rate of $\approx\Theta(n^{1.2})$. RRR, Ramen, Ramen, Rao.

TABLE 2. SPACE REQUIRED FOR RAMEN, RAMEN, RAO AND MINIMUM SPANNING TREE-BASED COLOR CLASS ENCODINGS OVER DIFFERENT NUMBERS OF SAMPLES (SIZES IN GB) AND TIME AND MEMORY REQUIRED TO BUILD MINIMUM SPANNING TREE

			MST						
Data set	*No. of samples*	*RRR matrix*	*Total space*	*Parent vector*	*Delta vector*	*Boundary bit vector*	*Build time (hh:mm:ss)*	*Expected edge weight*	$\frac{\text{size}(MST)}{\text{size}(RRR)}$
Homo sapiens RNA-seq samples	200	0.42	0.15	0.08	0.06	0.01	0:05:42	2.42	0.37
	500	1.89	0.46	0.2	0.24	0.03	0:12:15	3.42	0.24
	1000	5.14	1.03	0.37	0.6	0.06	0:25:03	4.39	0.2
	2000	14.2	2.35	0.71	1.5	0.14	0:51:58	5.38	0.17
	5000	59.89	7.21	1.72	5.1	0.39	3:52:34	6.61	0.12
	10,000	190.89	16.28	3.37	12.06	0.86	10:17:42	7.68	0.085
BBB	2586	15.8	2.66	0.63	1.88	0.16	00:57:43	6.98	0.17
Escherichia coli strain reference genomes	5598	2.06	0.83	0.02	0.76	0.06	00:03:15	7.8	0.4

Central columns break down the size of individual MST components.
BBB, Blood, Brain, and Breast; MST, Minimum Spanning Tree; RRR, Ramen, Ramen, Rao.

Table 3. The Minimum Spanning Tree Construction Time for 1000 Experiments Using Different Number of Threads

No. of threads	*1*	*2*	*4*	*8*	*16*	*32*
Run time (hh:mm:ss)	02:47:08	01:38:26	01:02:42	00:31:57	00:22:00	00:14:17

Memory stays the same across all the runs.

whereas the new MST-based representation grows roughly at a rate of $\Theta(n^{1.2})$. This explains why the RRR-based representation grows to dwarf the CQF (which grows roughly linearly) and become the bottleneck to scaling to larger data sets, whereas the MST-based representation does not. With the MST-based representation, the CQF itself is now the bottleneck.

Finally, the last two rows in Table 2 show the size of the RRR- and MST-based color-class representations for the human BBB and *Escherichia coli* data sets, respectively. BBB is the data set used in SBT and its subsequent tools (Solomon and Kingsford, 2017; Sun et al., 2017; Yu et al., 2018), as well as in Mantis (Pandey et al., 2018), and *E. coli* is the data set analyzed in the Rainbowfish article. This data set, which has been obtained from GenBank (O'Leary et al. 2016), consists of 5598 distinct *E. coli* strains. Since the strain assemblies are all from the same species, *E. coli*, each strain shares a large portion of its sequence with the others. We specifically chose this data set since Rainbowfish has already demonstrated a large improvement in size for it compared to Vari (Muggli et al., 2017).

As the table shows, our MST-based color-class representation is able to effectively compress genomic color data in addition to RNA-seq color data.

3.2.2. Index building evaluation. The ''Build time'' column in Table 2 shows the time required to build our MST-based color-class representation from Mantis' RRR-based representation. All builds used 16 threads. Table 3 shows how the MST construction time for a 1000 sample data set scales as a function of the number of build threads. The memory consumption is not affected by number of threads and remains fixed for all trials. The memory usage for both the main Mantis build and the MST construction steps is shown in Table 4. Since these phases are run independently, and since the MST phase follows the Mantis construction phase, the peak memory for the whole build pipeline is the maximum of the memory required for each of the two construction phases.

Overall, the MST construction time is only a tiny fraction of the overall time required to build the Mantis index from raw fastq files. The vast bulk of the time is spent processing the fastq files to produce filtered squeakrs. This step was performed on a cluster of 150 machines over roughly 1 week. Thus MST construction represents <1% of the overall index build time. The memory required to build the MST is dependent on the size of the CQF and grows proportional to that. In fact, due to the multi-pass construction procedure, the peak MST construction memory is essentially the size of the CQF plus a relatively small (and adjustable) amount of working memory. For the run over $10k$ experiments, where the CQF size was the largest ($98G$), the peak memory required to build MST is $111G$.

Table 4. The Memory Required for Mantis Build and Minimum Spanning Tree Compression Phases on Human RNA-Seq Data

Data set	*No. of samples*	*Mantis build memory (GB)*	*MST build memory (GB)*
H. sapiens RNA-seq samples	200	5	8
	500	10	16
	1000	18	29
	2000	25	29
	5000	58	59
	10,000	111	111
BBB	2586	28	29

The overall memory required to construct the full index is the max of the two columns which, for these datasets, is always the MST memory.

TABLE 5. QUERY TIME AND RESIDENT MEMORY FOR MANTIS USING THE MINIMUM SPANNING TREE-BASED REPRESENTATION FOR COLOR INFORMATION AND THE ORIGINAL MANTIS (USING RAMEN, RAMEN, RAO-COMPRESSED COLOR CLASSES) OVER 10,000 EXPERIMENTS

	Mantis with MST			*Mantis*		
	Index load+query	*Query*	*Space*	*Index load+query*	*Query*	*Space*
10 Transcripts	1 minute 10 seconds	0.3 seconds	118 GB	32 minutes 59 seconds	0.5 seconds	290 GB
100 Transcripts	1 minute 17 seconds	8 seconds	119 GB	34 minutes 33 seconds	11 seconds	290 GB
1000 Transcripts	2 minute 29 seconds	79 seconds	120 GB	46 minutes 4 seconds	80 seconds	290 GB

The ''query'' column provides just the time taken to execute all queries (as would be required if the index was already loaded in e.g., a server-based search tool). Note that, in resident memory usage for the MST-based representation, the counting quotient filter always dominates the total required memory.

3.2.3. Query evaluation. We evaluate query speed in the following manner. We select random subsets, of increasing size, of transcripts from the human transcriptome and query the Mantis index to determine the set of experiments containing each of these transcripts. Mantis answers transcript queries as follows. For each k-mer in the transcript, it computes the set of samples containing that k-mer. It then reports a sample as containing a transcript if the sample contains more than Θ fraction of the k-mers in the transcript, where Θ is a user-adjustable parameter. Note that, for Mantis, the Θ threshold is applied at the very end. Mantis first computes, for each sample, the fraction of k-mers that occur in that sample and then filters as a last step. Thus the query times reported here are valid for any Θ.

Table 5 reports the query performance of both the RRR and MST-based Mantis indexes. Despite the vastly-reduced space occupied by the MST-based index and the fact that the color class decoding procedure is more involved, query in the MST-based index is slightly faster than querying in the RRR-based index. The average query time in both RRR-based and MST-based index is 0.08 s/query.

Once the index has been loaded into RAM, Mantis queries are much faster than the three SBT-based large-scale sequence search data structures, and our MST-based color-class representation doesn't change that.

4. DISCUSSION AND CONCLUSION

We have introduced a novel exact representation of the color information associated with the cdbg. Our representation yields large improvements in terms of representation size compared to previous state-of-the-art approaches. While our MST-based representation is much smaller, it still provides rapid query and can, for example, return the query results for a transcript across an index of $10,000$ RNA-seq experiments in $\sim$0.08 s/query. Furthermore, the size benefit of our proposed representation over that of previous approaches appears to grow with the number of color classes being encoded, meaning it is not only much smaller but also much more scalable. Finally, the representation we propose is, essentially, a stand-alone encoding of the cdbg's associated color information, making this representation conceptually easy to integrate with any tool or method that needs to store color information over a large dbg.

Although it is not clear how much further the color information can be compressed while maintaining a lossless representation, this is an interesting theoretical question. It may be fruitful to approach this question from the perspective suggested by Yu et al. (2015), of evaluating the metric entropy, fractal dimension, and information-theoretic entropy of the space of color classes. Practically, however, we have observed that, at least in our current system, Mantis, for large-scale sequence search, the CQF, which is used to store the topology of the dbg and to associate color class labels with each k-mer, has become the new scalability bottleneck. In this study, it may be possible to reduce the space required by this component by making use of some of the same observations we relied upon to allow efficient color class neighbor search. For example, because many adjacent k-mers in the dbg share the same color class ID, it is likely possible to encode this label information sparsely across the dbg, taking advantage of the coherence between topologically nearby k-mers. Furthermore, to allow scalability to truly-massive data sets, it will likely be necessary to make the system hierarchical or even to adopt a more space-efficient (and domain-specific) representation of the underlying dbg. Nonetheless, because we have designed our color class representation

as essentially orthogonal to the dbg representation, we anticipate that we can easily integrate this approach with improved representations of the dbg.

Mantis with the new MST-based color class encoding is written in C++17 and is available at https://github.com/splatlab/mantis

AUTHOR DISCLOSURE STATEMENT

The authors declare they have no competing financial interests.

FUNDING INFORMATION

This work was supported by the U.S. National Science Foundation grants BIO-1564917, CSR-1763680, CCF-1439084, CCF-1716252, CNS-1408695, National Institutes of Health grants R01HG009937, R01GM122935, and Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative Grant GBMF4554. The experiments were conducted with equipment purchased through NSF CISE Research Infrastructure Grant Number 1405641. R.P. is a co-founder of Ocean Genomics.

REFERENCES

Alipanahi, B., Kuhnle, A., and Boucher, C. 2018a. Recoloring the colored de Bruijn graph, 1–11. *In International Symposium on String Processing and Information Retrieval*. Springer, Cham.

Alipanahi, B., Muggli, M. D., Jundi, M., et al. 2018b. Resistome SNP calling via read colored de Bruijn graphs. *bioRxiv*, 156174. Technical Report 312, Operation Research, NC state university, Waden, Germany.

Almodaresi, F., Pandey, P., and Patro, R. 2017. Rainbowfish: A succinct colored de Bruijn graph representation. *In LIPIcs-Leibniz International Proceedings in Informatics*, volume 88. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Oxford University Press, Oxford, England.

Almodaresi, F., Sarkar, H., Srivastava, A., et al. 2018. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics* 34, i169–i177.

Althaus, E., Funke, S., Har-Peled, S., et al. 2005. Approximating k-hop minimum-spanning trees. *Oper. Res. Lett.* 33, 115–120.

Bookstein, A., and Klein, S. 1991. Compression of correlated bit-vectors. *Inf. Syst.* 16, 387–400.

Bowe, A., Onodera, T., Sadakane, K., et al. 2012. Succinct de Bruijn graphs, 225–235. *In International Workshop on Algorithms in Bioinformatics*. Springer, Cham.

Bradley, P., den Bakker, H., Rocha, E., et al. 2017. Real-time search of all bacterial and viral genomic data. *bioRxiv* 234955.

Burrows, M., and Wheeler, D.J. 1994. A block-sorting lossless data compression algorithm.

Chang, Z., Li, G., Liu, J., et al. 2015. Bridger: A new framework for de novo transcriptome assembly using RNA-seq data. *Genome Biol* 16, 30.

Chikhi, R., Limasset, A., Jackman, S., et al. 2014. On the representation of de Bruijn graphs, 35–55. *In International Conference on Research in Computational Molecular Biology*. Springer, Cham.

Chikhi, R., and Rizk, G. 2012. Space-efficient and exact de Bruijn graph representation based on a Bloom filter, 236–248. *In International Workshop on Algorithms in Bioinformatics*. Springer, Cham.

Crawford, V., Kuhnle, A., Boucher, C., et al. 2018. Practical dynamic de Bruijn graphs. *Bioinformatics* 34, 4189–4195.

Elias, P. 1974. Efficient storage and retrieval by content and address of static files. *J ACM (JACM)* 21, 246–260.

Grabherr, M.G., Haas, B.J., Yassour, M., et al. 2011. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nat. Biotechnol.* 29, 644–652.

Holley, G., Wittler, R., and Stoye, J. 2016. Bloom filter trie: An alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.* 11, 3.

Iqbal, Z., Caccamo, M., Turner, I., et al. 2012. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.* 44, 226—232.

Khuller, S., Raghavachari, B., and Young, N.E. 2002. Balancing minimum spanning and shortest path trees. *CoRR* cs.DS/0205045.

Liu, B., Guo, H., Brudno, M., et al. 2016a. deBGA: Read alignment with de Bruijn graph-based seed and extension. *Bioinformatics* 32, 3224–3232.

Liu, J., Li, G., Chang, Z., et al. 2016b. Binpacker: Packing-based de novo transcriptome assembly from rna-seq data. *PLOS Comput. Biol.* 12, e1004772.

Manyem, P., and Stallmann, M.F.M. 1996. *Some Approximation Results in Multicasting*. Technical Report 312, Operation Research, NC state university, Raleigh, NC.

Marathe, M.V., Ravi, R., Sundaram, R., et al. 1998. Bicriteria network design problems. *CoRR* cs.CC/9809103.

Muggli, M.D., Bowe, A., Noyes, N.R., et al. 2017. Succinct colored de Bruijn graphs. *Bioinformatics* 33, 3181–3187.

Mustafa, H., Schilken, I., Karasikov, M., et al. 2019. Dynamic compression schemes for graph coloring. *Bioinformatics* 35, 407–414.

NIH. 2017. SRA. Available at: www.ebi.ac.uk/ena/browse Accessed November 6, 2017.

O'Leary, N.A., Wright, M.W., Brister, J.R., et al. 2016. Reference sequence (refseq) database at NCBI: Current status, taxonomic expansion, and functional annotation. *Nucleic Acids Res.* 44, D733–D745.

Ottaviano, G., and Venturini, R. 2014. Partitioned Elias-Fano indexes, 273–282. *In Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. ACM, New York City.

Pandey, P., Almodaresi, F., Bender, M.A., et al. 2018. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Syst.* 7, 201–207.e4.

Pandey, P., Bender, M.A., Johnson, R., et al. 2017a. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, i133–i141.

Pandey, P., Bender, M.A., Johnson, R., et al. 2017b. A general-purpose counting filter: Making every bit count, 775–787. *In Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, New York City.

Pandey, P., Bender, M.A., Johnson, R., et al. 2017c. Squeakr: An exact and approximate k-mer counting system. *Bioinformatics* 34, 568–575.

Pevzner, P.A., and Tang, H. 2001. Fragment assembly with double-barreled data. *Bioinformatics* 17(suppl_1), S225–S233.

Pevzner, P.A., Tang, H., and Waterman, M.S. 2001. An eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci U S A* 98, 9748–9753.

Raidl, G.R. 2008. *Exact and Heuristic Approaches for Solving the Bounded Diameter Minimum Spanning Tree Problem*. Vienna University of Technology, PhD Thesis.

Raman, R., Raman, V., and Rao, S.S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, 233–242. *In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. ACM, New York City.

Salikhov, K., Sacomoto, G., and Kucherov, G. 2014. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms Mol. Biol.* 9, 2.

Schulz, M.H., Zerbino, D.R., Vingron, M., et al. 2012. Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics* 28, 1086–1092.

Simpson, J.T., Wong, K., Jackman, S.D., et al. 2009. Abyss: A parallel assembler for short read sequence data. *Genome Res.* 19, 1117–1123.

Solomon, B., and Kingsford, C. 2016. Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.* 34, 300–302.

Solomon, B., and Kingsford, C. 2017. Improved search of large transcriptomic sequencing databases using split sequence bloom trees, 257–271. *In International Conference on Research in Computational Molecular Biology*. Springer, Cham.

Sun, C., Harris, R.S., Chikhi, R., et al. 2017. Allsome sequence bloom trees, 272–286. *In International Conference on Research in Computational Molecular Biology*. Springer, Cham.

Turner, I., Garimella, K.V., Iqbal, Z., et al. 2018. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics* 34, 2556–2565.

Yu, Y., Liu, J., Liu, X., et al. 2018. SeqOthello: Querying RNA-seq experiments at scale. *Genome Biol.* 19, 167.

Yu, Y.W., Daniels, N.M., Danko, D.C., et al. 2015. Entropy-scaling search of massive biological data. *Cell Syst.* 1, 130–140.

Zerbino, D.R., and Birney, E. 2008. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* 18, 821–829.

Ziv, J., and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 337–343.

Address correspondence to:
Fatemeh Almodaresi
Department of Computer Science
University of Maryland
College Park, MD 20742-5031

E-mail: falmodar@umd.edu rob@cs.umd.edu