ABSTRACT

Title of Dissertation: ALGORITHMS AND DATA STRUCTURES FOR INDEXING, QUERYING, AND ANALYZING LARGE COLLECTIONS OF SEQUENCING DATA IN THE PRESENCE OR ABSENCE OF A REFERENCE

Fatemeh Almodaresi
Doctor of Philosophy, 2020

Dissertation Directed by: Professor Rob Patro
Department of Computer Science

High-throughput sequencing has helped to transform our study of biological organisms and processes. For example, RNA-seq is one popular sequencing assay that allows measuring dynamic transcriptomes and enables the discovery (via assembly) of novel transcripts. Likewise, metagenomic sequencing lets us probe natural environments to profile organismal diversity and to discover new strains and species that may be integral to the environment or process being studied. The vast amount of available sequencing data, and its growth rate over the past decade also brings with it some immense computational challenges. One of these is how do we design memory-efficient structures for indexing and querying this data. This challenge is not limited to only raw sequencing data (i.e. reads) but also to the growing collection of reference sequences (genomes, and genes) that are assembled from this raw data. We have developed new data structures (both reference-based and reference-free)

to index raw sequencing data and assembled reference sequences. Specifically, we describe three separate indices, "Pufferfish", an index over a set of genomes or transcriptomes, and "Rainbowfish" and "Mantis" which are both indices for indexing a set of raw sequencing data sets. All of these indices are designed with consideration of support for high query performance and memory efficient construction and query.

The Pufferfish data structure is based on constructing a compacted, colored, reference de Bruijn graph (ccdbg), and then indexing this structure in an efficient manner. We have developed both sparse and dense indexing schemes which allow trading index space for query speed (though queries always remain asymptotically optimal). Pufferfish provides a full reference index that can return the set of references, positions and orientations of any k-mer (substring of fixed length "k") in the input genomes. We have built an alignment tool, Puffaligner, around this index for aligning sequencing reads to reference sequences. We demonstrate that Puffaligner is able to produce highly-sensitive alignments, similar to those of Bowtie2, but much more quickly and exhibits speed similar to the ultrafast STAR aligner while requiring considerably less memory to construct its index and align reads.

The Rainbowfish and Mantis data structures, on the other hand, are based on reference-free colored de Bruijn graphs (cdbg) constructed over raw sequencing data. Rainbowfish introduces a new efficient representation of the color information which is then adopted and refined by Mantis. Mantis supports graph traversal and other topological analyses, but is also particularly well-suited for large-scale sequence-level search over thousands of samples. We develop multiple and successively-refined versions of the Mantis index, culminating in an index that adopts a minimizer-

partitioned representation of the underlying k-mer set and a referential encoding of the color information that exploits fast near-neighbor search and efficient encoding via a minimum spanning tree. We describe, further, how this index can be made incrementally updatable by developing an efficient merge algorithm and storing the overall index in a multi-level log-structured merge (LSM) tree. We demonstrate the utility of this index by building a searchable Mantis, via recursive merging, over 10,000 raw sequencing samples, which we then scale to over 15,000 samples via incremental update. This index can be queried, on a commodity server, to discover the samples likely containing thousands of reference sequences in only a few minutes.

ALGORITHMS AND DATA STRUCTURES FOR INDEXING, QUERYING, AND ANALYZING LARGE COLLECTIONS OF SEQUENCING DATA IN THE PRESENCE OR ABSENCE OF A REFERENCE

by

Fatemeh Almodaresi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Rob Patro, Chair/Advisor
Professor Stephen Mount, University Chair Representative
Professor David Mount
Professor Hector Corrada Bravo
Professor Mihai Pop

Dedication

I dedicate this work to Khanoom, my grandma, that if it wasn't for her I couldn't image myself be the person I am now or better said, even be!

Acknowledgments

"And whatever of blessings and good things you have, it is from Allah." Quran (16:53)

There are many people to whom I owe my gratitude for helping me during my PhD years and making this thesis a piece of work that I will be thrilled about forever.

First and foremost, I would like to express my deepest appreciation to my advisor, Professor Patro. I started my work with him as someone who "believed" is incapable of any achievements and if it wasn't for his profound belief in my work and my abilities and for his patience and relentless support I would not be where I am now. He has not only guided me to become a critical thinker and a professional scientist but also a better and stronger person. Taking his class was the pivotal moment of my life that changed it entirely. I now see myself as a proficient researcher who is now able and passionate to work on novel problems and he is my role model for all of that. I have watched him and learned a lot from his extensive knowledge in his area of expertise, persistence in accomplishing the task, and his constructive advice and ingenious contributions. The work in this thesis was impossible without him.

I had great pleasure of collaborating with many researchers and scientists in the field during my PhD and I thank them all for the precious experience they offered.

I would like to thank all the members of Mantis project, specially Michael Ferdman and Robert Johnson for fruitful conversations and insightful suggestions during the three years of collaboration and Prashant Pandey for his invaluable contribution.

I would like to thank my thesis committee members for all of their guidance through this process; your discussion, ideas, and feedback have been absolutely invaluable.

Many thanks to my colleagues in our lab and CBCB that had made my PhD journey full of fun and exciting. Special thanks to Laraib Iqbal Malik who was like a sister to me and I miss her deeply as well as Mohsen Zakeri for being the brother that was always there for me. I would also like to acknowledge help and support from some of the staff members, specifically Kathy Germana, Tom Hurst, and Barbara Lewis for their kindness, patience and guidance for all the administrative tasks with a constant smile always on their faces.

I cannot begin to express my thanks to my husband, Hamid, for his unwavering support, encouragement and accommodation throughout the past years. He gave up the life back in our country and his career for me to pursue my dream which has now become a reality. He has my deepest appreciation for never giving up on me. I would also like to extend my deepest gratitude to my parents for the love, support, and constant encouragement I have gotten over the years. I want to specifically thank my mother who has gone through a lot of struggle and pain but has never surrender nurturing and supporting me. In the end, many thanks to my little siblings, specially Gazelle for her relentless support.

Finally, I am extremely thankful to all my friends and family for their per-

manent and unfailing support. Specifically, Leila Khatami, who is always my fuel and motivator. Her help cannot be overestimated as she was always there for me throughout the hard time when I was alone and far from family. Also, my deepest appreciation goes to Zohreh Sadeghi and Shohreh Tabesh whom never wavered in their support. I would like to specifically thank our Iranian and religion community in NY, who were the most welcoming people one could imagine. Our friends at Stony Brook always make us feel at home and our friends in NY enriched us with a lot of knowledge and love that could not have been achieved otherwise. I was the luckiest person to have them in my life and will always owe a great part of my mindset, and perspective to them.

Funding: I gratefully acknowledge support from NSF grants BBSRC-NSF/BIO-1564917, IIS-1247726, IIS-1251137, CNS- 1408695, CCF-1439084, CCF-1617618, CCF-1716252, and from Sandia National Laboratories. The following grants specifically funded the Mantis line of work discussed in chapters 5 and 6: National Science Foundation grants CSR- 1763680, CCF-1439084, CCF-1716252, CCF-1750472, CNS-1408695, CNS-1763680 National Institutes of Health grants R01HG009937, and R01GM122935. The experiments were conducted with equipment purchased through NSF CISE Research Infrastructure Grant Number 1405641.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALSOME-SBT	All-Some Sequence Bloom Tree
BV	Bit Vector
CCDBG	Compacted Colored de Bruijn Graph
CDBG	Colored de Bruijn Graph
CIGAR	Concise Idiosyncratic Gapped Alignment Report
CQF	Counting Quotient Filter
CSEQ	Contig Sequence
CTAB	Contig Table
DBG	de Bruijn Graph
EQ	Equivalence Classes
ETAB	Extension Table
FN	False Negative
FP	False Positive
MEM	Maximally Extended Match
MMP	Maximum Mappable Prefix
MPHF	Minimum Perfect Hash Function
MST	Minimum Spanning Tree
NCBI	National Center for Biotechnology Information
PCQF	Partitioned Counting Quotient Filter
POS	Position Table
SBT	Sequence Bloom Tree
SRA	Sequence Read Archive
SSBT	Split Sequence Bloom Tree
TN	True Negative
TP	True Positive
UNI-MEM	Unique Maximally Extended Match

Chapter 1: Introduction

Modern RNA-seq protocols, driven by short-read sequencing by synthesis techniques, produce tens of millions or more short reads per sample, and have become a great asset for tasks such as transcriptome abundance estimation and assembly. The data produced by these protocols and made publicly-available comprises a rich database of information over hundreds of thousands of individuals samples, with associated meta-data and extensive variation. In other words, this well-established and low-cost protocol for producing short reads has provided the community with a massive collection of raw sequence samples. In addition to the large collections of raw sequence samples, there exist large databases of assembled genomes, metagenomes and transcriptomes. There has been a lot of effort and research in organizing and indexing sequence databases and extracting information from them efficiently [125]. To extract and analyze this information, a dynamic and efficient search index, with fast queries, over such big databases is essential. In addition to all these applications, these databases themselves are a great resource of information to find out sequence-based novelties, differences, disparities and abnormalities across different species, individuals, tissues, or even among single cells.

1 Reference-based and Reference-free Indexing

The inherent differences in the properties of raw (short-read) sequence samples and assembled reference sequences necessitate different computational strategies and brings about two different lines of work called reference-based and reference-free indexing respectively. A reference-based index is defined as an index over a collection of assembled sequences (typically longer than thousands of bases). Such indices have been used in various computational and biological applications, such as to perform alignment and mapping for genome and transcriptome abundance estimation, or to compare a sample to a reference to discover or catalog variation. A reference-free index, on the other hand, is an index over the collection of raw sequencing reads themselves.

1.1 Reference-based Indices

Aligning and mapping sequence reads to a reference genome or transcriptome is an important and unavoidable step of many pipelines in genome and transcriptome analysis. For almost all types of quantification and gene and RNA sequence expression analyses, we first need to align short reads to the reference transcript. However, in many analyses, this step is a time-consuming bottleneck. To speed up the alignment process, researchers have developed seed-and-extend methodologies to first find an exact match to a seed from read and continue aligning from that point. Many popular indices are used for the seed-and-extend approach including k-mer-based indices used in tools such as [89], full-text self indices such as the FM-index

used in Bowtie2 [79] and the FMD-index used in BWA-MEM [83], and the suffix array-based index used in tools such as STAR [39]. There have been recent efforts to extend both approaches to the context of indexing different types of sequence graphs [125], with tradeoffs between space and time efficiency. On the succinct self-index side, one notable example is gramtools, the tool in which the graph itself is represented as a modified BWT [101]. For the recently developed k-mer lookup based approaches, however, it is more prevalent to use graphs as the underlying data structure. Tools like deBGA [95], genomeMapper [143], and BGREAT [92] are examples of such a methodology.

A wide variety of tools have been developed for indexing references and querying large collections of sequencing reads against them in the past [39, 57, 74, 79, 80, 83, 87, 88, 89]. One way we can divide these indices into two main categories of full-text and hash-based indices. In the full-text approaches, the series of sequences are put together and treated as one large text and then indexed based on well-known data structures for indexing large-text sequences such as FM-index or suffix array [39, 74, 79, 80, 83, 87]. These indices are usually very small, compared to the size of the raw data they index but can be slow to query. On the other hand, those based on indexing fixed-length patterns of size k (k-mers) by putting them in a hash table are quite fast to query but grow large quickly. Therefore, having an index with a balance between the memory and query time is still a continuing computational challenge. Reference-based indices are the main focus of chapters 2 and 3. In chapter 2, we introduce Pufferfish [6] as a reference-based indexing scheme which couples a practically (and asymptotically) fast hash-based scheme with moderate memory

encoding by pairing a graph-based representation for the list of sequences with minimum perfect hash functions (MPH) [93]. In chapter 3 we expand the Pufferfish index to a full short-read aligner that produces highly-accurate results compared to the most sensitive aligners in the field such as Bowtie2 [79] and STAR [39] in much less time than Bowtie2 and using much less memory than STAR.

1.2 Reference-free Indices

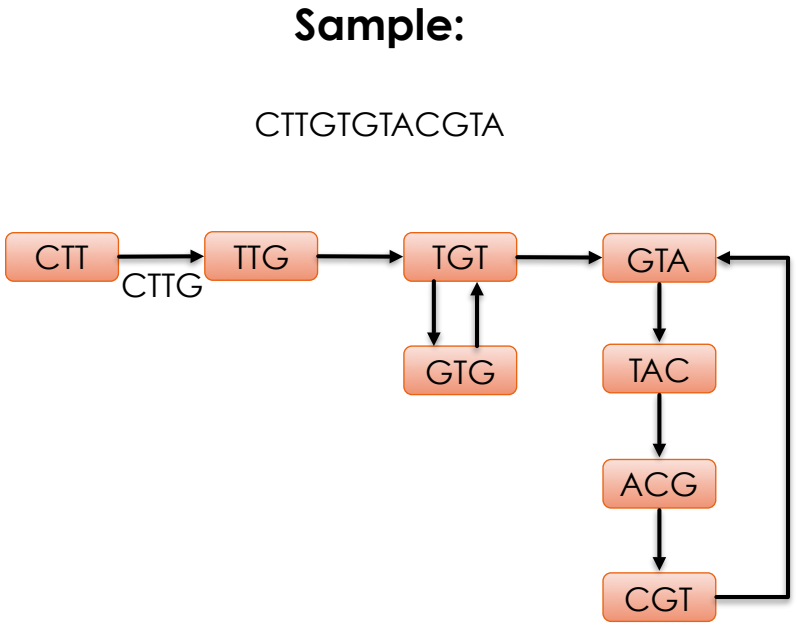
There is also a different line of work focused on building various types of indices over the raw short-read data to solve a problem commonly called "large-scale sequence search." The main property of raw sequence databases that makes them fundamentally different from assembled sequences is that the sequences are short, incomplete, unprocessed (and thus contain artifacts and contaminants) and highly-redundant. Specifically, this redundancy grows extensively when dealing with the short sequences in the form of collections of k-mers. Graph-based indices make use of factoring out the repeats in the sequences to reduce the size of the index and provide faster queries, and thus are particularly suitable for this type of data. In these approaches, a sequence is split into sub-sequences of size k (called a k-mer) where each k-mer is presented only once in the index. One of the mostly common types of sequence graphs is the de Bruijn graph and its variants, the colored de Bruijn graph (cdbg), the compacted de Bruijn graph, and the compacted colored de Bruijn graph. I have worked on two particular problems in this domain. The first is to develop a succinct representation of membership information in a colored

de Bruijn graph, and the second is an efficient indexing for k-mers in a compacted colored de Bruijn graph. Below, I provide a brief overview of previous works on graph-based representation, along with common use cases of the various algorithms and methods developed.

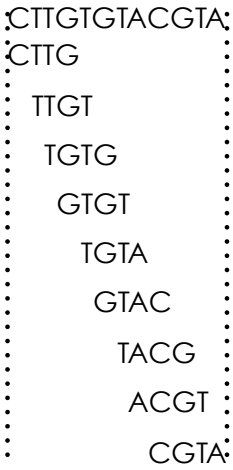
2 De Bruijn Graph

A de Bruijn graph (dbg) is a directed graph representing a set of sequences. This type of graph has two variants, node-centric and edge-centric. In the edge-centric de Bruijn graph, each directed edge is a unique substring of length k in the sequence set, which we call a k-mer. Each edge has a prefix overlap of $k-1$ bases with the source node and a suffix overlap of length $k-1$ with the destination node [125]. Figure 1a shows a simple de Bruijn graph for a sample with one string. This type of graph is designed so that by having a walk through edges and putting all edges next to each other with overlaps of $k-1$, we are able to build the reference sequence, such as a gene or transcript, as shown in 1b. Of course it is worth noting that such "perfect assembly" is not always possible due to sequencing errors, repeats, and other complexities that arise. In the node-centric variant of a de Bruijn graph, each node represents a k-mer, and the adjacency relationship is defined by overlapping prefixes and suffixes in the same manner.

The de Bruijn graph is a useful representation of a reference or set of sequencing reads that helps faster assessment of the sequence similarity in biological tasks such as assembly or variation detection.



(a) de Bruijn graph for a sample with one sequence.



(b) (k+1)-mers and assembled sequence retrieved walking the de Bruijn graph.

Figure 1: Building a de Bruijn graph and reconstructing the reference sequence from it. This example shows how one can reconstruct the reference sequence having a walk through nodes and edges in a de Bruijn graph and taking care of overlaps.

One drawback is that for the $k-1$ overlaps between consequent edges, the obvious data structures to store this graph are very space-inefficient. For example, ABYSS [146] represents the de Bruijn graph as a hash table with each k-mer as the key and a byte keeping all the connections to other nodes as its value. It needs 1 bit to show the existence of each of the edges in the forward or reverse-complement direction (as we have four characters in our alphabet, we can expand the current node to reach to the next one in at most four different ways in forward direction). The space such a data structure takes is $|E_s|\,(\frac{k}{4}+1)\frac{1}{\gamma}$ bits where $\gamma \leq 1$, is the hash table loading factor. This storage is large for even one moderately-sized genome data set, such as the human genome (starting from 40GB and depending on the loading factor it can grow to 100GB or more). Yet, a few different data structures and algorithms have been proposed to reduce the size of a de Bruijn

graph and represent it efficiently. One category of these data structures use the Bloom Filter [16] to represent a de Bruijn graph [28, 29, 63, 128, 138]. Also, there are a few proposed representations that rely on succinct data structures [52] and rank and select operations including the original work by Conway and Bromage [35] and later the work in [18] that is called the BOSS representation of de Bruijn graph from the authors' initials. BOSS is an efficient, edge-centric representation of de Bruijn graph that takes around 3 bits per k-mer, which is considerably smaller than the hash table representation. This representation provides a mechanism for navigation through the de Bruijn graph and also an interface to interact and get access to the ID of each k-mer. In section 4, I explain in more detail, our work on the color representation for a de Bruijn graph built on top of the BOSS structure, using the interface it provides.

2.1 Compacted de Bruijn graph

The main advantage of a colored compacted de Bruijn graph is being more space-efficient compared to the classic representation of the de Bruijn graph due to the nodes representing paths with no branches rather than k-mers. The process of compacting the de Bruijn graph is meant to merge all k-mers in a non-branching path in the de Bruijn graph with outgoing and incoming degrees of one into a single node which is called a *unitig*. The output of this step is called a compacted de Bruijn graph, that connects these unitigs. It is a variant of the original de Bruijn graph with unitigs as nodes rather than the k-mers. This can be used in the same way as a

de Bruijn graph for different downstream applications, such as mapping, alignment, variant detection, etc. This method reduces memory by eliminating the potentially large amount of overlaps of $k-1$ bases repeated in consequent k-mers. For instance, the output node after merging two consequent nodes in a node-centric de Bruijn graph with overlap of $k-1$ would be a unitig of length $k+1$ where the node starts with the first base in the source node, continues with the $k-1$ overlapping bases and ends in the last base of the destination. This compaction step can greatly reduce the memory required to represent the graph and is very useful in cases where we are dealing with repeat-heavy sequences [95]. Recently, researchers have designed and implemented algorithms for building a colored compacted de Bruijn graph directly from raw data instead of building the memory-inefficient de Bruijn graph first and then compacting it [31, 105]. However, indexing a colored compacted de Bruijn graph is still a challenge that needs further investigation.

A number of tools exist that use the de Bruijn graph as an index for various purposes including kallisto [22], deBGA [95], deSALT [96] that are used for indexing compacted de Bruijn graphs. While most of these indices provide fast query, the memory they need is large, so that in the case of large datasets their memory requirements become impractical or intractable. In section 2, we propose a memory-efficient indexing data structure for a colored compacted de Bruijn graph that has an asymptotically constant (and pratically fast) expected k-mer lookup time. We first use TwoPaCo to build the colored compacted de Bruijn graph [105], and then develop a novel data structure to index such compacted de Bruijn graphs while

keeping a balance between space and query time. One great advantage of a specific variant of our indexing structure (sparse indexing) is the flexibility that it provides by giving the option of trading time for space by means of a tunable parameter.

2.2 Colored de Bruijn graph

A colored de Bruijn graph is a generalized form of a de Bruijn graph that allows representing multiple samples in one unified graph while keeping the identity of (and information specific to) each sample [66]. The samples may be the result of different experiments for the same species, known variants of the same sequence, or different sequencing samples. By counting all of the samples together as one and building a de Bruijn graph from them, we will lose information about the variations happening across samples. Colored de Bruijn graphs were originally proposed by Iqbal et. al [66] in a tool named *cortex*, useful for variant discovery and genotyping. Each sample is represented with a unique color in a colored de Bruijn graph and hence all the k-mers coming from that sample will carry that color with them. To be exact, each k-mer or edge in a colored de Bruijn graph has a color set showing all the samples that this k-mer has appeared in. Maintaining each color separately, we can differentiate between bubbles that are induced by *repeats* when we see the coverage evenly distributed along different paths from those induced by *errors* where one side of the branch has a much lower coverage [66]. There are other data structures to represent colored de Bruijn graphs as well, implemented in tools such as BFT [63] and VARI [109]. However, such data structures, the color information itslef is the

dominant part in the total space the colored de Bruijn graph takes compared to the small portion that is taken by the de Bruijn graph representation.

In chapter 4 we propose a succinct data structure to represent colors in a colored de Bruijn graph paired with any de Bruijn graph representation that provides a unique index for each k-mer. We prove the succinctness of our data structure and compare our space and query time results with VARI, which uses a similar API (and the same de Bruijn graph data structure, BOSS) to construct the index and find bubbles in the colored de Bruijn graph.

3 Sequence Search

The ability to issue sequence-level searches over publicly available databases of assembled genomes and known proteins has played an instrumental role in many studies in the field of genomics, and has made BLAST [11] and its variants some of the most widely-used tools in all of science. However, these indices are defined over a database of reference sequences. Yet, the vast majority of publicly-available sequencing data (e.g., the data deposited in the SRA [76]) exists in the form of raw, unassembled sequencing reads for which the reference-based indices are not a suitable choice. Such indices are unsuitable, first, because they do not scale well as the amount of data grows to the size of the SRA (which today is ≈ 4 petabases of sequence information) and second, because relatively long queries (e.g., genes) are unlikely to be present in their entirety as an approximate substring of the input in the raw sequence reads (which are usually less than 200 nucleotides long).

This problem was first introduced and tackled by Solomon and Kingsford [149]. They introduced a data structure that enables an efficient type of search over thousands of sequencing experiments. Specifically, they re-phrase the query and each separate experiment of reads in terms of k-mer set membership in a way that is robust to the fact that the target sequences have not been assembled. The resulting problem is coined as the *experiment discovery* problem, where the goal is to return all experiments that contain at least some user-defined fraction θ of the k-mers present in the query string. The space and query time of the SBT structure has been further improved by [150] and [153]. However, scaling this representation is still an issue which leads us to the next tool that we've worked on called Mantis.

In chapter 5, we introduce Mantis, a space- and time-efficient index for searching sequences in large collections of experiments which is based on colored de Bruijn graphs. The "color" associated with each k-mer in a colored de Bruijn graph is the set of experiments in which that k-mer occurs. We use an exact counting quotient filter [121], an Approximate Membership Query (AMQ) structure to store a table mapping each k-mer to a color ID, and another table mapping color IDs to the actual set of experiments containing that k-mer. We achieve 20% times smaller memory footprint, and up to $108X$ improvement in query time compared to the split sequence bloom tree representation [150]. We also describe how we reduce the size of the index even further by developing a new encoding of the color information representation in an improved variant of the Mantis index that we call MST-based Mantis.

The MST-based Mantis data structure is able to scale to $10,000$ or more raw sequencing samples on a moderate server. However, scaling the index further remains a challenge and the main bottleneck preventing further scaling is the large counting quotient filter holding the k-mer to color identifier mapping. In chapter 6, we tackle this problem by partitioning the counting quotient filter into smaller blocks based on the k-mer minimizers. This partitioning scheme is particularly effective in sequencing data, as the index of the counting quotient filter partition for each k-mer is self-contained and thus there is no need to an additional structure. We also enable updatability for Mantis by incorporating the MST-based Mantis index into an LSM-tree structure [120, 136, 137]. This feature is important since it allows an existing index to be updated rather than requiring the index be rebuilt as new samples are added. We develop a methodology for merging two MST-based Mantis indices as prerequisite for the LSM-tree. The details of the memory-and-time aware merging process is explained in chapter 6.

4 Overview of this document and contribution

I start the document by introducing indices on databases of reference-based sequences. In the next chapter, chapter 2, I cover the details of data structures used for indexing and querying in a database of long assembled sequences and the challenges specific to this type of data. I talk about the index structure we designed in Pufferfish [6] for indexing a set of assembled genomes or transcriptomes. In chapter 3 I explain how we develop a seqeuence aligner around the Pufferfish index. In

chapter 4, I switch to reference-free indices and remain on that subject for the rest of the document. I explain in detail the new color representation we introduced for the k-mers' color information in a colored compacted de Bruijn graph in our tool, Rainbowfish. In chapter 5, I go over the data structure we present in the Mantis paper for indexing a colored compacted de Bruijn graph combining the previously introduced representation of colors from Rainbowfish and the counting quotient filter filter for mapping the k-mers to their corresponding color ID. In chapter 6, I describe the algorithmic and engineering improvements we make to be able to scale Mantis to a larger number of samples. I also cover the design that enables the updatability feature for the index to support dynamic insertion of the new samples. Finally, I give a short summary of my PhD journey, my accomplished projects and results, and potential future extensions or utilization of the results in the Conclusion, chapter 7.

Contribution:

- Chapter 2: Joint work with Hirak Sarkar. I contributed equally in the design, coding and writing.
- Chapter 3: Joint work with Mohsen Zakeri. In the coding I contributed in writing the sections related to mapping and chaining as well as filling gaps in the alignment. In the experiments, I mainly contributed in designing the metagenomic experiemnts (last two sections of the results). We both participated equally in writing.
- Chapter 4: I wrote almost all the code and ran the experiments. Other con-

tributors helped with the writing.

- Chapter 5: This is a work with the Mantis team. All the brainstorming and development of the ideas have happened throughout our weekly brainstorming sessions with almost all the members participating. I was responsible for majority of the coding, and testing as well as running all the experiments. I also participated in some sections of the writing.

- Chapter 6: This is a work with the Mantis team. All the brainstorming and development of the ideas have happened throughout our weekly brainstorming sessions with almost all the members participating. I was responsible for most (more than 80%) of the implementation, and testing as well as running all the experiments. I was also the main writer and others helped with revising the text in the chapter.

Chapter 2: Pufferfish: A space and time-efficient compacted de Bruijn graph index [6]*

1 Introduction

Motivated by the tremendous growth in the availability and affordability of high-throughput genomic, metagenomic and transcriptomic sequencing data, the past decade has seen a large body of work focused on developing data structures and algorithms for efficiently querying large texts (e.g. genomes or collections of genomes) [39, 57, 74, 79, 80, 83, 87, 88, 89]. While numerous approaches have been proposed, many fall into one of two categories — those based on indexing fixed-length pattern occurrences (i.e., k-mers, which are patterns of length k) in the reference sequences [57, 88, 89] (most commonly using hashing), and those based on building full-text indices such as the suffix array or FM-index over the references [39, 74, 79, 80, 83, 87].

Recently, there have been efforts to extend both categories of approaches from the indexing of linear reference genomes to the indexing of different types of sequence graphs [125], with various tradeoffs in the resulting space and time efficiency. On the full-text index side, examples include approaches such as those of Maciuca

*A joint work with Hirak Sarkar published in ISMB2018

et al. [101] and Beller and Ohlebusch [14] which encode the underlying graph using variants of the BWT, and the approach of Sirén [147], which indexes paths in the variation graph (again making use of a substantially modified BWT). There have also been recent approaches based on k-mer-indices that adopt graphs as the underlying representation of the text being searched. Examples of such tools include genomeMapper [143], BGREAT [92], Kallisto [22] and deBGA [95].

Rather than general variation graphs, we focus in this manuscript on the de Bruijn graph. The de Bruijn graph is a widely-adopted structure for genome and transcriptome assembly [54, 56, 130]. However, the compacted variant of the de Bruijn graph has recently been gaining increasing attention both as an indexing data structure—for use in read alignment [95] and pseudoalignment [22]—as well as a structure for the analysis of variation (among multiple genomes) [105] and a reference-free structure for pan-genome storage [63]. The colored compacted de Bruijn graph [30, 106, 107] (see Section 2 below) is particularly attractive for representing and indexing repetitive sequences, since exactly repeated sequences of length at least k are represented only once in the set of unique, non-branching paths. As has been demonstrated by Liu et al. [95], this considerably speeds up alignment to repeat-heavy genomes (e.g., the human genome) as well as to collections of related genomes. Here, we consider collections of genomes to be represented as color information on the de Bruijn graph (as described by Iqbal et al. [66]; see Section 2 below for details). Efficient representation of multiple references encoded as colors in a de Bruijn graph has been investigated in tools such as VARI [109] and Rainbowfish [5]. Both VARI and Rainbowfish have implemented a data structure to efficiently index

color encoding on top of a succinct navigational representation of a de Bruijn graph, proposed in BOSS [18]. However none of these tools are equipped with membership queries and sequence search and are, hence, regarded as out of scope in the present paper.

The query speed of existing colored compacted de Bruijn graph indices comes at a considerable cost in index size and memory usage. Specifically, the need to build a hash table over the k-mers appearing in the de Bruijn graph unipaths requires a large amount of memory, even for genomes of moderate size. Typically, these hash functions map each k-mer (requiring at least 8 bytes) to the unipath in which it occurs (typically 4 or 8 bytes) and the offset where the k-mer appears in this unipath (again, typically 4 or 8 bytes). A number of other data structures are also required, but, most of the time, this hash table dominates the overall index size. For example, an index of the human genome constructed in such a manner (i.e., by deBGA or kallisto) may require 40—100GB of RAM (see Table 2). This already exceeds the memory requirements of moderate servers (e.g., those with 32G or 64G of RAM), and these requirements quickly become untenable with larger genomes or collections of genomes.

2 Preliminaries

In this brief section, we formally define the preliminary terms and notations that are used throughout the manuscript. We consider all strings to be over the alphabet $\Sigma = \{A, C, G, T\}$. A k-mer is a string of length k over Σ (i.e. $k \in \Sigma^k$). Given a

k-mer, x, we define the reverse complement of x by $\bar{x}$; this is a string obtained by reversing x and then complementing each character according to the rule $\bar{A} = T, \bar{C} = G, \bar{G} = C, \bar{T} = A$. We define the canonical representation of a k-mer, x, by $\hat{x} = \min(x, \bar{x})$, where the minimum is taken according to the lexicographic ordering. In this manuscript, we are fundamentally interested in indexing a collection of reference sequences (be they pre-existing, or assembled *de novo*); we therefore adopt the following definitions with respect to the de Bruijn graph and its variants. The de Bruijn graph is a graph, $G = (V, E)$, built over the k-mers of some reference string, s. We define $s(k)$ as the set of k-mers present in s, and assume that s is of length at least k (i.e. $|s| \geq |k|$). The vertex set of G is given by $V = \{\hat{x} \mid x \in s(k)\}$. There exists an edge $\{u, v\} \in E$ between two vertices u and v if and only if there exists some $(k+1)$-mer, z, in S such that u is a prefix of z and v is a suffix of z. The colored de Bruijn graph associates each $v \in V$ with some specific set of colors. When building the de Bruijn graph over a collection of reference strings $s_1, \ldots, s_M$, we define the color set for a vertex to be the set of references in which it appears (i.e. $\text{colors}(v) = \{i \mid v \in s_i(k) \vee \bar{v} \in s_i(k)\}$). Finally, we define the compacted colored de Bruijn graph to be the *color-coherent* compaction of a colored de Bruijn graph. A compacted de Bruijn graph replaces each non-branching path, $p = u \rightsquigarrow v$, in G with a single edge (which no longer represents a single k-mer, but instead represents the entire string that would be spelled out by walking from u to v in an orientation consistent manner). We say that such a compaction is *color-coherent* if and only if all vertices $u \in p$ share the same color set. The compacted colored de Bruijn graph is the graph obtained by performing a maximal color-coherent compaction of the

colored de Bruijn graph.

3 Method

We present Pufferfish, a software tool implementing a novel indexing data structure for the colored compacted de Bruijn graph and the colored colored compacted de Bruijn graph. We focus on making the colored compacted de Bruijn graph index practical in terms of disk and memory resources for genomic and metagenomic data while maintaining very fast query speeds over the index. While we are conscious of memory usage, we don't aim to build the smallest possible index. Furthermore, we introduce two different variants of our index, the *dense* and *sparse* Pufferfish indices. Similar to the FM-index [47], in the *sparse* Pufferfish index, there is a sampling factor that can be tuned to trade off search speed for index size. The dense index is, in a sense, just a variant of the sparse index tuned for maximum speed (and, hence, taking maximum space). However, as we believe the dense index will be a popular choice, we implement a few optimizations and describe the structures separately.

Pre-processing We assume as input to Pufferfish the colored compacted de Bruijn graph on the reference or set of references to be indexed. The Pufferfish software itself accepts as input a graphical fragment assembly (GFA) format* file that describes the colored compacted de Bruijn graph. Specifically, this file encodes the unipaths (i.e., non-branching paths) of the colored compacted de Bruijn graph as "segments" and the mapping between these unipaths and the original reference sequences as

*https://github.com/GFA-spec/GFA-spec

"paths". Each path corresponds to an input reference sequence (e.g., a genome), and is spelled out by an ordered set of unipath IDs and the orientation with which these unipaths map to the reference, so that each unipath has an overlap of $k-1$ with its following unipath in the path (either in the forward or reverse-complement direction).

GFA is an evolving standard that is meant to be a common format used by tools dealing with graphical representations of genomes or collections of genomes. We note that there are a number of software tools for building the colored compacted de Bruijn graph directly (i.e., without first building the un-compacted de Bruijn graph). We adopt TwoPaCo [105], which employs a time and memory-efficient parallel algorithm for directly constructing the colored compacted de Bruijn graph, and whose output can be easily converted into GFA format. We note that, due to a technical detail concerning how TwoPaCo constructs the colored compacted de Bruijn graph and the GFA file, the output cannot be directly used by Pufferfish. Therefore, the current workflow of Pufferfish includes a GFA-to-GFA converter that prepares the TwoPaCo-generated GFA file for indexing by Pufferfish. We note that TwoPaCo (and therefore Pufferfish) consider the edge-explicit de Bruijn graph. That is, two k-mers will be connected if and only if the input reference contains a $(k+1)$-mer having one of these k-mers as its left k-mer and the other as its right k-mer. Conversely, other tools, like BCALM2 [31] and Kallisto consider the induced-edge de Bruijn graph, where there will be an edge between any pair of k-mers overlapping by $k-1$ nucleotides, regardless of whether or not a $(k+1)$-mer containing them exists in the input. This leads to small but persistent differences in the topology of

these graphs.

3.1 The dense Pufferfish index

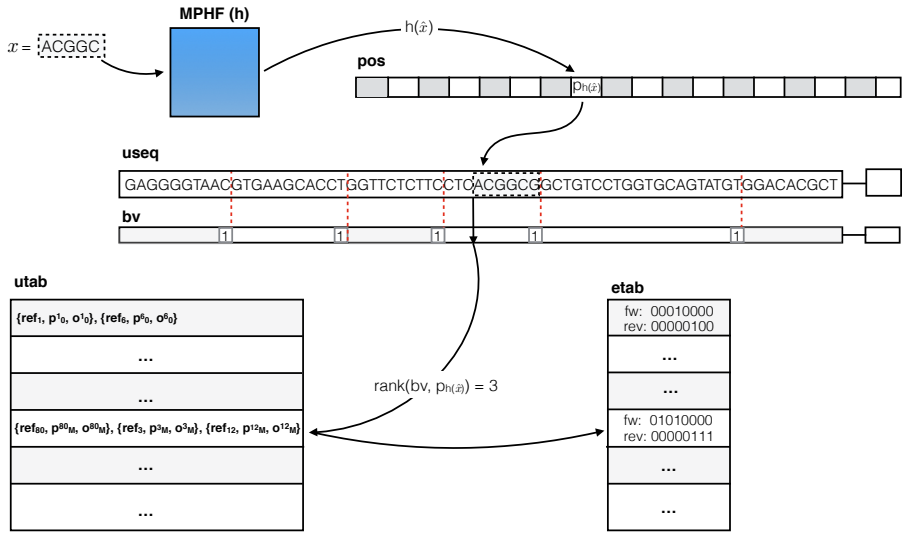


Figure 2: An illustration of searching for a particular k-mer, x, in the *dense* Pufferfish index. The minimum perfect hash yields the index, $p_{\mathrm{h}(\hat{x})}$ in the `pos` vector where the k-mer appears in the unipath array. The k-mer is validated against the sequence recorded at this position in `cseq` (and, in this case, it matches). A rank operation on $p_{\mathrm{h}(\hat{x})}$ is performed in the boundary vector (`bv`), which yields the corresponding unipath-level information in the unipath table (`ctab`). If desired, the relative position of the k-mer within the unipath can be retrieved with an extra select and rank operation. Likewise, the rank used to determine this unipath's `ctab` entry can also be used to look up the edges adjacent to this unipath in the `etab` table if desired.

The index consists of 6 components (and an optional 7^{th} component), and the overall structure is similar to what is explained by Liu et al. [95]. Here, we provide a detailed description of the components of the dense Pufferfish index:

`cseq`: The unipath sequence array (`cseq`) consists of the (2-bit encoded) sequence of all unipaths of the colored compacted de Bruijn graph packed together into a single array. Typically, the size of this structure is close to (or smaller than)

the size of the 2-bit encoded reference sequence, since redundant sequences are represented only once in this structure. We note that the unipath array contains the sequence of every valid k-mer, as well as that of potentially invalid k-mers (those which span unipath boundaries in the packed array, as the sequences in the array follow each other without any delimiters or gaps.). We denote by L_s the total length (in nucleotides) of the unipath array.

`bv`: The boundary vector (`bv`) is a bit-vector of length L_s. The bits of this vector are in one-to-one correspondence with the nucleotides of the unipath array, and the boundary vector contains a one at each nucleotide corresponding to the end of a unipath in `cseq`, and a zero everywhere else. We can retrieve the index of each unipath in `cseq` using the RANK operation on `bv`. RANK(`bv`, i) returns the number of 1s in `bv` before the current index, i, or, in other words, the index of the current unipath. This can be used to get reference information for the current unipath from `ctab`, which is explained below. We note that `bv` is typically *very* sparse, and so can likely be compressed (using e.g., RRR [132] or Elias-Fano encoding), though we have not explored this yet.

h : The minimum perfect hash function (h) maps every *valid* k-mer in the unipath array (i.e., all k-mers not spanning unipath boundaries) to a unique number in $[0, N)$, where N is the number of distinct valid k-mers in `cseq`. We make use of the highly-scalable minimum perfect hash function (MPHF) construction algorithm of Limasset et al. [93]. We also note that we build the MPHF on the canonicalized version of each k-mer.

`pos`: The position vector (`pos`) stores, for each valid k-mer x, the position where this k-mer occurs in `cseq`. Specifically, for k-mer x, let $\bar{x}$ be the reverse complement of x and let $\hat{x}$ be the canonical form of x (the lexicographically smaller of x and $\bar{x}$). Then $\texttt{pos}\left[\mathrm{h}(\hat{x})\right]$ contains the starting position of x in `cseq` such that $\texttt{cseq}\left[\mathrm{h}(\hat{x}) : \mathrm{h}(\hat{x}) + k\right] = x$.

`ctab`: The unipath table (`ctab`) stores, for each unipath appearing in `cseq`, the reference sequences (including reference ID (**ref**), offset (**p**) and orientation (**o**) in Fig. 2) where this unipath appears in the reference. This is similar to a "posting list" in traditional inverted indices, where all occurrences of the item (in this case, an entire colored compacted de Bruijn graph unipath) are listed. The order of the unipaths in `ctab` is the same as their order in `cseq`, allowing the information for a unipath to be accessed via a simple rank operation on `bv`.

`etab`: The edge table (`etab`) stores, for each unipath appearing in `cseq`, the nucleotides that encode the edges to the left and right of this unipath. The edge table maintains a byte for each unipath, where each byte encodes which of the left and right extensions of this unipath produce a valid k-mer in the de Bruijn graph. Specifically, the first four bits of the byte are set to 1 if there is a left neighbor that can be reached by taking the leftmost $(k-1)$-mer of the current unipath and pre-pending `A`, `C`, `G`, and `T` respectively, and these bits are 0 otherwise. The last 4 bits of the byte likewise encode the connectivity for the right end of the unipath. This edge table is useful for speeding up

navigation in the graph, because we find that the compacted de Bruijn graph is often *sparse*, so that querying for all potential neighbors of a unipath can be wasteful, since many unipaths have few neighbors.

`eqtab`: *Optionally*, an equivalence class table that records, for each unipath, the set of reference sequences where this unipath appears. Pre-computation and storage of these equivalence classes can speed up certain algorithms (e.g., pseudoalignment [22]).

These structures allow us to index every k-mer in the colored compacted de Bruijn graph efficiently, and to recall, on demand, all of the reference loci where a given k-mer occurs. We note here that the k-mers of the colored compacted de Bruijn graph constitute only a subset of the k-mers in `cseq`. We refer to all k-mers in `cseq` that do not span the boundary between two unipaths as *valid* k-mers; these are in one-to-one correspondence with the k-mers of the colored compacted de Bruijn graph.

Additionally, we note that navigation among the unipaths in the index could be accomplished without an explicit edge table. Specifically, upon reaching the end of a unipath, one could query the index with all possible extensions to see which are supported by the indexed sequence, and potentially spurious overlaps (i.e., unipaths which overlap by $k-1$ nucleotides but are not actually adjacent in any reference sequence) can be filtered out by traversing the relevant entries of `ctab`. However, this process is not efficient, and is particularly wasteful if the average degree of each unipath is small since, in this case, most queries for neighbors would fail or

return spurious overlaps which would then be filtered out. An empirical analysis of the compacted colored de Bruijn graph of the datasets we analyze suggested that these graphs do, in fact, tend to have a skewed degree distribution, and that most unipaths exhibit a small degree. This motivates the utility of `etab`, especially given that it takes relatively small space.

3.1.1 k-mer query in the dense Pufferfish index

By using a minimum perfect hash function (MPHF), h, to index the *valid* k-mers, we avoid the typically large memory burden associated with standard hashing approaches. Instead, the identity of the hashed keys is encoded implicitly in `cseq`. Given a k-mer x, we can check for its existence and location in the following way. We first compute $i = h(\hat{x})$, the index assigned to the canonicalized version of k-mer x by h. If $i \geq N$, where N is the number of unique *valid* k-mers, then we immediately know that x is not a valid k-mer. Otherwise, we retrieve the position p_i stored in $\texttt{pos}[i]$. Finally, we check if the encoded string $\texttt{cseq}[p_i : p_i + k]$ is identical to x (or $\bar{x}$). If so, we have found the unipath location of this k-mer. Otherwise, x is not a valid k-mer. Here, we use the notion $S[i : j]$ to mean the substring of S from index i (inclusive) to index j (exclusive) with length $j - i - 1$.

Given p_i, we can retrieve the reference positions by computing $r_{p_i} = \text{RANK}(\texttt{bv}, p_i)$, which provides an index into `ctab` that is associated with the appropriate unipath. This provides all of the reference sequences, offsets and orientations where this unipath appears. We compute the offset of k-mer x in the unipath as $o_i =$

$p_i - \text{SELECT}(r_{p_i})$, where $\text{SELECT}(r_{p_i})$ returns the start position of the unipath in `ctab`. This allows us to easily project this k-mer's position onto each reference sequence where it appears. We note that querying a k-mer in the Pufferfish index is an asymptotically constant-time operation, and that the reference loci for a k-mer x can be retrieved in $\mathcal{O}(\texttt{occ}(x))$ time, where $\texttt{occ}(x)$ is the number of occurrences of x in the reference.

3.2 The sparse Pufferfish index

The Pufferfish index, as described above, is relatively memory-efficient. Yet, what is typically the biggest component, the `pos` vector, can still grow rather large. This is because it requires $\lceil \lg(|\texttt{cseq}|) \rceil$ bits for each of the N valid k-mers in `cseq`. However, at the cost of a slight increase in the practical (though not asymptotic) complexity of lookup, the size of this structure can be reduced considerably. To see how, we first make the following observation:

Observation 1. *In the colored compacted de Bruijn graph (and hence, in* `cseq`*), each valid k-mer occurs exactly once (k-mers occuring between unipath boundaries are not considered). Hence, any valid k-mer in the colored compacted de Bruijn graph is a complex k-mer (i.e., it has an in or out degree greater than 1), a terminal k-mer (i.e., it appears at the beginning or end of some input reference sequence), or it has a unique predecessor and / or successor in the orientation defined by the unipath.*

We can exploit this observation in Pufferfish to allow *sampling* of the k-mer

positions. That is, rather than storing the position of each k-mer in the unipath array, we store the position only for some subset of k-mers, where the rate of sampling is given by a user-defined parameter s. For those k-mers that are not sampled, we store, instead, three pieces of information; the extension that must be applied to move toward the closest k-mer at a sampled position (the `QueryExt` vector), whether or not the corresponding k-mer in `cseq` is canonical (the `isCanon` vector), and whether the extension to reach the nearest sampled position should be applied by moving to the right or the left (the `Direction` vector). The `QueryExt` vector encodes the extensions in a 3-bit format so that variable-length extensions can be encoded, though every entry in this vector is reserved to take the same amount of space (3 times the maximum extension length, e). The `isCanon` vector is set to 1 whenever the corresponding k-mer appears in `cseq` in the canonical orientation, and is set to 0 otherwise. The `Direction` vector is set to 1 whenever the corresponding, non-sampled, k-mer should be extended to the right, and it is set to 0 when the corresponding k-mer should be extended to the left. We additionally store an extra bit vector with the same size as `cseq` (the `isSamp` vector) that is set to 1 for any k-mer whose position is sampled and 0 for all other k-mers.

This idea of sampling the positions for the k-mers is similar to the idea of sampling the suffix array positions that is employed in the FM-index [47], and the idea of walking to the closest sampled position to verify a k-mer occurs is closely related to the shallow forest covering idea described by Belazzougui et al. [13] for verifying membership of a k-mer in their fully-dynamic variant of the de Bruijn graph. This scheme allows us to trade off query time for index space, to allow the

Pufferfish index to better scale to large genomes or collections of genomes.

3.2.1 k-mer query in the sparse Pufferfish index

k-mer query in the sparse Pufferfish index is the same as that in the dense index, except for the first step — determining the position of the k-mer x in `cseq`. When we query the MPHF with x to obtain $i = h(\hat{x})$, there are three possible results.

1. In the first case, if $i \geq N$, this implies, just as in the dense case, that x is not a valid k-mer.

2. In the second case, if $i < N$ and $\texttt{isSamp}[i] = 1$, this implies that we have explicitly stored the position for this k-mer. In this case we can retrieve that position as $p_i = \texttt{pos}[\text{RANK}(\texttt{isSamp}, i)]$ and proceed as in the dense case to validate x and retrieve its reference positions.

3. In the third case, if $i < N$ and $\texttt{isSamp}[i] = 0$, this implies we do not know the position where x would occur in `cseq`, and we must find the closest sampled position in order to decode the position of x (if it does, in fact, occur in `cseq`). This is accomplished by Algorithm 1.

Intuitively, Algorithm 1 appends nucleotides stored in the `QueryExt` array to x to generate a new k-mer, x', which either has a sampled position, or is closer to a sampled position than is x. The extension process is repeated with x', x'', etc. until either an invalid position is returned by h, or a sampled position is reached. If an invalid position is returned at any point in the traversal, the original k-mer cannot

Algorithm 1 Find Query Offset

```
procedure FindQueryOffset
    x ← the query k-mer
    x̂_q ← x̂
    i ← h(x̂)
    offset ← 0
    while i < N and not isSamp[i] do
        extIdx ← i-rank(isSamp, i)
        extNuc ← QueryExt [extIdx]
        extLen ← len(extNuc)
        if isCanon[extIdx] and Direction[extIdx] then
            x ← x̂[extLen :] + extNuc
            offset ← offset + e
        end if
        if not isCanon[extIdx] and Direction[extIdx] then
            x ← x̄̂[extLen :] + extNuc
            offset ← offset + e
        end if
        if isCanon[extIdx] and not Direction[extIdx] then
            x ← extNuc + x̂[: −extLen]
            offset ← offset − e
        end if
        if not isCanon[extIdx] and not Direction[extIdx] then
            x ← extNuc + x̄̂[: −extLen]
            offset ← offset − e
        end if
        i ← h(x̂)
    end while
    if i ≥ N then return −1
    end if
    p_i ← pos [rank(isSamp,i)] - offset
    if cseq[p_i : p_i + k] == x̂_q or cseq[p_i : p_i + k] == x̄̂_q then
        return p_i
    else
        return −1
    end if
end procedure
```

have been a valid query. On the other hand, if a sampled position is reached, one still needs to verify that the k-mer implied by the query procedure is identical to the original k-mer query x (or $\bar{x}$). To check this, one simply traverses back to the position in cseq for the original k-mer x that is implied by the sampled position and sequence of extension operations. The rest of the search proceeds as for the dense case. The whole process of a (successful) k-mer query in sparse index is illustrated in Figure 3 through an example.

By altering the stored extension size e and the maximum sampling rate s, one can limit the maximum number of extension steps (and hence the maximum number of hash lookups) that must be performed in order to retrieve the potential index of x in cseq. A denser sampling and longer extensions require fewer possible extension steps, while a sparser sampling and shorter extensions require less space for each non-sampled position. If $e \geq \frac{s-1}{2}$, one can guarantee that at most a single extension step needs to be performed for any k-mer query, which allows k-mer queries to remain practically very fast while still reducing the index size for large reference sequences.

Even though the sparse index maintains a number of extra bit vectors not required by the dense index, it is usually considerably smaller. Assume a case where the extension length $e = \frac{s-1}{2}$ is approximately half of the sampling factor (the minimum length that will guarantee each query requires at most a single extension step). Since we keep the extension required to get to the closest position in the left or right direction, we need to keep e bases for a k-mer, with each base represented using 3 bits (since we need to allow encoding extensions of length $< e$, for which

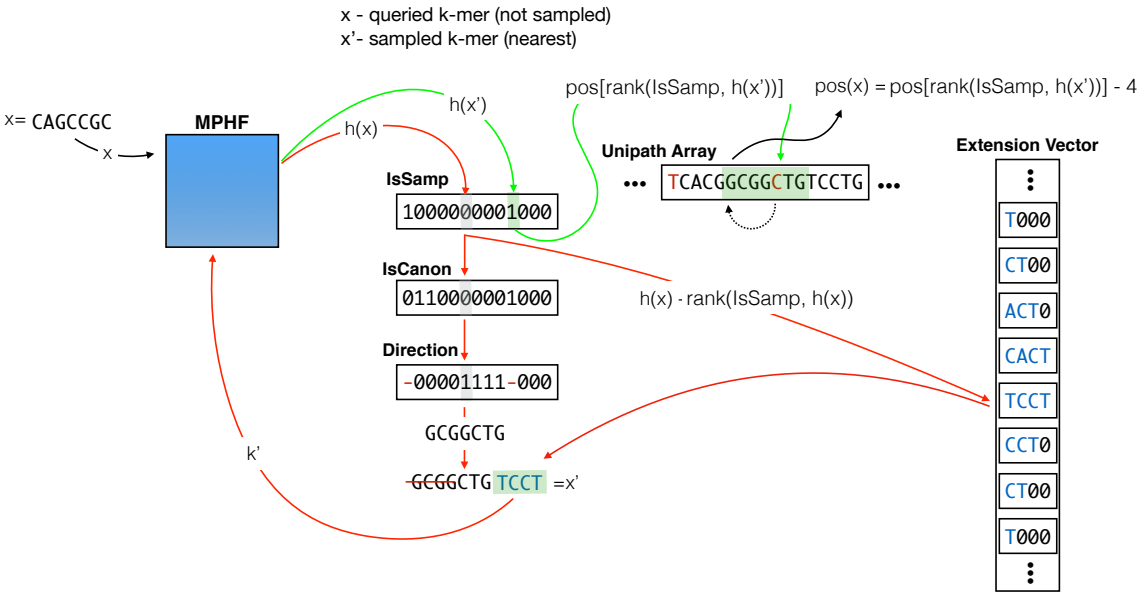


Figure 3: An illustration of searching for a particular k-mer in the *sparse* pufferfish index with sample factor (s) of 9 and extension size (e) of 4. Vector `isSamp` has length equal to the number of valid k-mers, and `isCanon` and `Direction` have length equal to the total number of non-sampled k-mers. The minimum perfect hash yields the index $\mathrm{h}(\hat{x})$ for $x =$ `CAGCCGC` in `isSamp`, where we discover that the k-mer's position is not sampled. Since `isCanon` $[\mathrm{h}(\hat{x}) - \textsc{rank}(\texttt{isSamp}, \mathrm{h}(\hat{x}))] = 0$ we know that the k-mer, if present, is not in the canonical orientation in `cseq`. Since x *is* in the canonical orientation, we must reverse-complement it as $\bar{x} =$ `GCGGCTG` before adding the extension nucleotides. Then, based on the value of `Direction`$[\mathrm{h}(\hat{x}) - \textsc{rank}(\texttt{isSamp}, \mathrm{h}(\hat{x}))]$, we know that to get to the closest sampled k-mer we need to append the extension nucleotides to the right of $\bar{x}$. The extension is extracted from the `QueryExt` vector. Since extensions are recorded only for non-sampled k-mers, to find the index of the current k-mer's extension, we need to determine the number of non-sampled k-mers preceding index $\mathrm{h}(\hat{x})$. This can easily be computed as $\mathrm{h}(\hat{x}) - \textsc{rank}(\texttt{isSamp}, \mathrm{h}(\hat{x})$, which is the index into `QueryExt` from which we retrieve this k-mers's extension. We create a new k-mer, x', by appending the new extension to $\bar{x}$, and also removing its first $e = 4$ bases. Then, we repeat the same process for the new k-mer x'. This time, the k-mer is sampled. Hence, we go directly to the index in `cseq` suggested by `pos`$[\textsc{rank}(\texttt{isSamp}, \mathrm{h}(\hat{x'}))]$. To check if the original k-mer we searched for exists, we need to compare the k-mer starting from $e = 4$ bases to the left of the current position with the *non-canonical* version of the original k-mer (since the sampled k-mer x' was arrived at by extending the original query k-mer by 4 nucleotides to the right). Generally speaking, once we reach a sampled position, to check the original query k-mer, we need to move in `cseq` to either the right or the left by exactly the distance we traversed to reach this sample, but in the opposite direction.

the encoding must allow a delimiter). Hence, this requires $3e$ bits per k-mer for the `QueryExt` vector. The `isCanon` and `Direction` vectors each require a single bit per non-sampled k-mer, and the `isSamp` vector requires a single bit for all N of the valid k-mers. Assume, for simplicity of analysis, that the sampled k-mers are perfectly evenly-spaced (which is not possible in practice since e.g., we must require to sample at least one k-mer from each unipath), so that the number of sampled k-mers is simply given by $\frac{N}{s} = \frac{N}{2e+1}$. Further, since we are ignoring unipath boundary effects, assume that $N = L_s$. Since the space required by the rest of the index components (e.g. the MPHF, and `ctab`, etc.) is the same for the dense and sparse index, the sparse index will lead to a space savings whenever $\frac{N}{2e+1}\lceil \lg(N) \rceil + \left[N + \left(N - \left(\frac{N}{2e+1}\right)\right)(3e+2)\right] < N\lceil \lg(N) \rceil$. Under this analysis, in a typical dataset, such as the human genome with $\lg(L_s) \approx \lg(N) \approx \lg(3 \times 10^9) \geq 30$ bits, and choosing $s = 9$ and $e = 4$, so that we sample every 9^{th} k-mer on average, and require at most one extension per query, we save, on average, ~ 14.5 bits per k-mer. Of course, the practical savings are less because of the boundary effects we ignored in the above analysis.

4 Evaluation

We explored the size of the index along with the memory and time requirements for index building and k-mer querying (a fundamental building block of many mapping and alignment algorithms) using Pufferfish and two other tools, BWA (BWA-MEM [83], specifically) and Kallisto.

Tool	Memory (MB)			Time (h:m:s)		
	Human Transcriptome	Human Genome	Bacterial Genomes	Human Transcriptome	Human Genome	Bacterial Genomes
BWA	292	4,443	32,213	0:02:56	0:58:27	13:11:45
Kallisto	3,552	150,657	315,387	0:03:05	3:27:42	9:07:35
pufferfish dense	1,466	27,438	75,342	0:04:13	2:09:25	13:10:00
pufferfish sparse	1,466	27,438	75,342	0:04:41	2:28:53	13:46:11
TwoPaCo	1,466	9,380	17,407	0:02:47	0:34:43	9:59:05
pufferize	584	27,438	75,342	0:0:10	0:21:53	1:03:17
pufferfish dense index	438	20,000	50,459	0:01:16	0:51:20	2:07:38
pufferfish sparse index	331	17,745	50,457	0:01:44	1:10:48	2:43:49

Table 1: Upper half of the table shows construction time and memory requirements for BWA, Kallisto and Pufferfish (dense and sparse) on three different datasets. In the lower half of the table, the construction statistics are provided for different phases of Pufferfish pipeline. The time requirement for Pufferfish is the sum of different sub parts of the workflow, where the memory requirement is the *max* of the same.

Though BWA is not a graph-based index, it was chosen as it implements the highly memory-efficient FMD-index [83], which is representative of a memory-frugal approach. It is also worth noting that, although we only test querying for fixed-length k-mers here, BWA is capable of searching for *arbitrary* length patterns — an operation not supported by the Kallisto or Pufferfish indices. On the other hand, Kallisto [22] adopts a graph-based index, and provides very fast k-mer queries. Both BWA and Kallisto implement all phases of index construction (i.e., the input to these tools is simply the FASTA files to be sequenced). For Pufferfish, however, we first need to build the colored compacted de Bruijn graph. We build the colored compacted de Bruijn graph and dump it in GFA format using TwoPaCo [105]. Then (as the output does not satisfy our definition of a colored compacted de Bruijn graph) we need to further prepare the GFA file for indexing. We call this process *pufferization*. It converts the GFA file to the format accepted by Pufferfish (i.e., each k-mer should appear only once in either orientation among all the unipaths, and all unipaths connected in the colored compacted de Bruijn graph should have an overlap of exactly

$k-1$ bases). Finally, we build both dense and sparse Pufferfish indexes and benchmark the time and memory for all steps of the pipeline individually. All experiments were performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD running ubuntu 16.10, and were carried out using a single thread except for colored compacted de Bruijn graph building step using TwoPaCo. For all datasets, we consider $k = 31$, and the sparse Pufferfish index was constructed with $s = 9$ and $e = 4$.

References and query datasets We performed benchmarking on three different reference datasets, selected to demonstrate how the different indices scale as the underlying reference size and complexity increases. Specifically, we have chosen a common human transcriptome (GENCODE version 25, 201 MB, having 79,334,030 distinct k-mers), a recent build of the human genome (GRCh38, 2.9 GB, having 2,652,229,049 distinct k-mers), and an ensemble of > 8000 bacterial genomes and contigs (18G, having 5,350,807,438 distinct k-mers) downloaded from RefSeq (`ftp://ftp.ncbi.nlm.nih.gov/genomes/refseq/bacteria/`). The human transcriptome represents a small reference sequence (which nonetheless exhibits considerable complexity due to e.g., alternative splicing), the human genome represents as a moderate (and very common) size reference, and the collection of bacterial genomes acts as a large reference set. For the k-mer query experiments, we search for all the k-mers from an experimental sequencing dataset associated with each reference. To query the human transcriptome, we use k-mers from SRA accession SRR1215997,

Tool	Memory (MB)			Time (h:m:s)		
	Human Transcriptome	Human Genome	Bacterial Genome	Human Transcriptome	Human Genome	Bacterial Genome
BWA	308	4,439	27,535	0:17:35	0:50:31	0:14:05
Kallisto	3,336	110,464	232,353	0:02:01	0:19:11	0:22:25
pufferfish dense	454	17,684	41,532	0:02:46	0:10:37	0:06:03
pufferfish sparse	341	12,533	30,565	0:08:34	0:22:11	0:08:26

Table 2: The time and memory required to load the index and query all k-mers in reads of the input FASTQ files for different datasets.

with 10,683,470 reads, each of length 100 bases. To query the human genome, we use k-mers from SRA accession SRR5833294 with 34,129,891 reads, each of length 76 bases. Finally, to query the bacterial genomes, we use k-mers from SRA accession SRR5901135 (a sequencing run of *E. coli*) with 2,314,288 reads of variable length.

Construction time The construction time for various methods depends, as expected, on the size and complexity of the references being indexed (Table 1). No tool exhibits faster index construction than all others across all datasets, and the difference in construction time between the fastest and slowest tools for any given dataset is less than a factor of 3. All tools perform similarly for the human transcriptome. For indexing the human genome, BWA is the fastest, followed by Pufferfish and then Kallisto. For constructing the index on all bacterial genomes, Kallisto finished most quickly, followed by BWA and then Pufferfish. The time (and memory) bottleneck of index construction for Pufferfish is generally TwoPaCo's construction of the colored compacted de Bruijn graph. This is particularly true for the bacterial genomes dataset where TwoPaCo's colored compacted de Bruijn graph construction accounts for $\sim 85\%$ of the total index construction time. This motivates considering potential improvements to the TwoPaCo algorithm for large collections of genomes

(as well as considering other tools which may be able to efficiently construct the required colored compacted de Bruijn graph input for Pufferfish).

Construction memory usage Unlike construction time, the memory required by the different tools for index construction follows a clear trend; BWA requires the least memory for index construction, followed by Pufferfish, and Kallisto requires the most memory. There are also larger differences in the construction memory requirements than the construction time requirements. For example, to construct an index on the human genome, Kallisto requires $\sim$ 34 times more memory than BWA (and $\sim$ 5.5 times more memory than Pufferfish). With respect to the current pipeline used by pufferfish, we see that TwoPaCo is the memory bottleneck for the human transcriptome and bacterial genomes datasets, while *pufferize* consumes the most memory for the human genome. For the bacterial genomes dataset in particular, TwoPaCo consumes over 3 times as much memory as the next most intensive step (*pufferize*) and $\sim$ 4.8 times as much memory as actually indexing the input colored compacted de Bruijn graph. We note that TwoPaCo implements a multi-pass algorithm, which can help control the peak memory requirements in exchange for performing more passes (and therefore taking longer to finish). However, we did not thoroughly explore different parameters for TwoPaCo's Bloom filter size (which indirectly affects the number of passes).

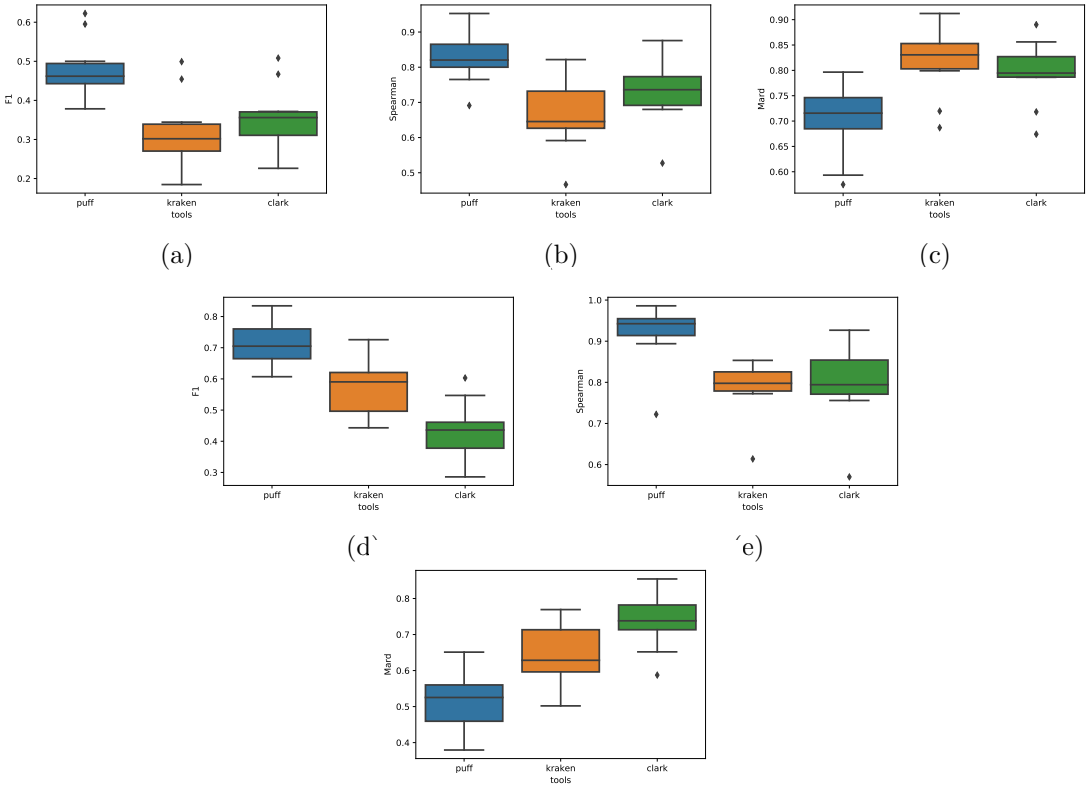


Figure 4: Full taxonomy classification evaluation for three tools of Kraken, Clark, and Pufferfish. In a, b, and c we compare the F-1, spearman correlation, and mean absolute relative difference (mard) metrics for the results of the three tools over the 10 simulated read datasets of LC1-8 and HC1,2 without using any filtering options. In the plots in the second row, we evaluate accuracy of reports after running each tool with their default filtering option. (which filters out any mapping with less than 20% kmer coverage for Kraken, 44 nucleotide coverage for Pufferfish and without a "high-confidence" for Clark.)

5 Applying the Pufferfish index to taxonomic read assignment

In addition to benchmarking index construction and the primitive lookup operations, we also decided to apply the Pufferfish index to a problem where we thought its characteristics might be useful. To this end, we implemented a prototype system for taxonomic read assignment based on Pufferfish and a minor modification of the kraken algorithm, described in the seminal work of Wood and Salzberg [160].

Specifically, we consider a Pufferfish index built over complete bacterial and archaeal genomes (this is Kraken's `bacteria` database), and we implement a lightweight mapping algorithm where, for each read, we seek a consistent (i.e. co-linear) chain of unique maximal exact matches (uni-MEMs [95]). To determine to which node in the taxonomy a read should be assigned, we adopt Kraken's basic algorithm with the following modification. Instead of scoring each root-to-leaf path based on the number of k-mers shared between the read and the taxa along the path, we consider the union of all the intervals of the read that are covered by consistent chains of uni-MEMs (i.e. number of nucleotides covered in the mapping). For example, consider a read r that has uni-MEM matches with respect to the genomes of two species s_1 and s_2, where the corresponding intervals of the read covered by matches to s_1 are $[i, j], [i', j']$ and with respect to s_2 are $[k, \ell], [k', \ell']$ such that the covered intervals on each genome are consistent (i.e., co-linear and nearby in the reference). In this case, we define the coverage score of the read with respect to s_1 to be $S(r, s_1) = |\{i, \ldots, j\} \cup \{i', \ldots, j'\}|$, and likewise for $S(r, s_2)$. Further, let g be the parent genus of s_1 and s_2. We define $S(r, g) = |\{i, \ldots, j\} \cup \{i', \ldots, j'\} \cup \{k, \ldots, \ell\} \cup \{k', \ldots, \ell'\}|$. This process is

repeated up to the root of the tree such that the score for any given node n is determined by the union of the covered intervals for the subtree rooted at n. Using this definition for the score, we then simply adopt Kraken's algorithm of assigning the read to the node with the highest-scoring root-to-leaf path (or assigning the read to the LCA of all such nodes in the case of ties.

The main potential benefit of this approach over the k-mer-based approach of kraken is that this notion enforces positional consistency among the substrings of the read and leaf taxa that are used as evidence of a match. Additionally, this approach favors greater coverage of the read instead of simply a larger shared k-mer count — a notion that we believe is likely to be more indicative of a good alignment when these measures disagree.

We implemented our prototype tool for taxonomic read assignment and benchmarked it against both kraken [160] and Clark [119]. We adopt a subset of the benchmarks, and simulated data (LC1-8, HC1, HC2) considered by McIntyre et al. [104]. The metrics under which we evaluate the tools are the Spearman correlation, MARD, and the F1 score. However, rather than considering these metrics at any specific taxonomic rank, which leads to the problem of how to evaluate false positives that are assigned at a different rank, we consider these metrics aggregated over the entire taxonomy. In this full-taxonomy evaluation, we consider the maximally specific predictions made by each method. Then, we recursively aggregate the counts up the taxonomy to higher ranks (such that a parent node receives the sum of the assigned reads of its children, plus any reads that were assigned directly to this node). The same aggregation was performed on the true counts.

This metric provides a single statistical evaluation, over the entire taxonomic tree, that prefers reads mapped (1) along the correct root-to-leaf path and (2) closer along this path to the true node of origin compared to assignments that are either on the wrong path entirely, or further from the true node of origin. In addition to this comprehensive measure, we provide further collection of different accuracy metrics on this data.

We evaluate the output of these tools in both their unfiltered modes (which assign any read with a single k-mer/ uni-MEM match between the query and reference) and using their default filtering criteria (where some score or confidence threshold must be attained before a read can be assigned to a taxon. The results depicted in Fig. 4 show that Pufferfish provides the best estimates under all metrics, followed by Clark in unfiltered mode and by kraken in filtered mode. We also consider the time and memory required by these tools to perform taxonomic read assignment on a real experimental dataset consisting of $\sim 100M$ reads.

6 Discussion & Conclusion

In this paper we proposed a new efficient data structure for indexing compacted colored de Bruijn graphs, and implement this data structure in a tool called Pufferfish. We showed how Pufferfish can achieve a balance between time and space resources. By building upon a MPHF [93], we provide practically fast k-mer lookup, and by carefully organizing our data structure and making use of succinct representations where applicable, we greatly reduce the space compared to traditional hashing-based

implementations. The main components of the data structures are a minimum perfect hash function (MPHF) built on k-mers, the concatenated unipath array from which the k-mers are sampled, a bit vector that marks the boundary of unitigs in the concatenated array, a vector containing the offset position for the k-mers, and a unipath table enumerating the occurrences of each unipath in the reference sequences.

Moreover, we presented two variants of the Pufferfish data structure; namely, a dense and a sparse variant. The first is optimized for fast queries and the second provides the user with the ability to trade off space for speed in a fine-grained manner. In the sparse index, we only keep offset positions for a subset of k-mers. To query a k-mer whose position is not sampled, the sparse representation is aided with a few auxiliary data structures of much smaller size. Since the largest component of the index is the position vector, adopting this sparse representation significantly reduces the required memory and disk space. Our analyses suggest that Pufferfish (dense) achieves similar speed to existing hash-based approaches, while greatly reducing the memory and disk space required for indexing, and that Pufferfish (sparse) reduces the required space even further, while still providing fast query capabilities. We consider indexing and querying on both small (human transcriptome) and large ($>$ 8000 bacterial genomes) reference datasets. Pufferfish strikes a desirable balance between speed and space usage, and allows for fast search on large reference sequences, using moderate memory resources.

Finally, we demonstrate the application of Pufferfish to the problem of taxonomic read assignment. We show that, using essentially the same algorithm as

kraken, Pufferfish can enable faster and more accurate taxonomic read assignment while using less memory. The accuracy benefit mostly results from replacing the k-mer-centric scoring of reads to taxa with a score based on the coverage of reads by taxa under consistent chains of uni-MEMs. This scoring scheme enforces positional consistency, and is enabled by the Pufferfish index. It more closely approximates a natural intuition of what it means for a read to match a taxon well, but can still be computed very efficiently.

Having built an index for a reference genome, transcriptome, or metagenome using Pufferfish, the immediate future work consists of implementing more relevant applications based on this index. Many of these applications fall into the categories of problems that need mapping or alignment as their initial step. In our prototype taxonomic read assignment system, we have already implemented a basic mapping procedure, and this could easily be extended into a selective-alignment-style algorithm [140] to provide true edit distances or edit scripts. An aligner based around the Pufferfish index could be used to quickly align against collections of transcripts and genomes, and this could be useful in downstream tasks, such as contaminant detection, metagenomic abundance estimation (related to but distinct from taxonomic read assignment), etc. Finally, we believe that having a single graph against which we can align reads that is capable of representing many sequences simultaneously will admit an efficient approach for the joint alignment of RNA-seq reads to both the genome and the transcriptome. We can construct a de Bruijn graph that contains both the reference genome as well as the annotated transcript sequences. Reads which are then well-explained by annotated transcripts can be aligned effi-

ciently and accurately, while the genomic sequence can simultaneously be searched for evidence of new splice junctions; potentially improving both the efficiency and accuracy of existing RNA-seq alignment methods. We expect the memory efficiency of Pufferfish will be beneficial in working with larger collections of genomic, transcriptomic, and metagenomic datasets.

Chapter 3: Puffaligner: A Fast, Efficient, and Accurate Aligner Based on the Pufferfish Index*

1 Introduction

Since its introduction, next generation sequencing (NGS) has been widely used as a low-cost and accessible technology to produce high-throughput sequencing reads for many important biological assays. The sequencing data that is generated in the form of short reads, drawn from longer molecular fragments, and finding the optimal alignments of these short reads to some reference is a necessary first step for many downstream biological analyses. The process of finding the segment on the reference that is most similar to the query read, and therefore most likely to be the source of the fragment from which the read was drawn, is known as read mapping or read alignment.

The main goal in read alignment is to find alignments of contiguous sub-string of the underlying reference that yields a minimum edit distance (or maximum alignment score) between the read and the reference sequence at the alignment position. If the reads are paired-end, characteristics other than the alignment score can be used to filter spurious alignment locations, such as orientation of each end of the

*A joint work with Mohsen Zakeri

alignment pair (forward or reverse) or distance between the alignments corresponding to reads that are ends of the same fragment.

Short-read aligners are a major workhorse of modern genomics. Given the importance of the alignment problem, a tremendous number of different tools have been developed to tackle this problem. Some widely used examples are BWA [86], Bowtie2 [78], Hisat2 [73, 75] and STAR [40]. Existing alignment tools use a variety of indexing methods. Some tools, such as BWA, Bowtie2, and STAR use a full-text index over the reference sequences; BWA and Bowtie2 use variants of the FM-index, while STAR uses a suffix array.

A popular alternative approach to full-text indices is to instead, index substrings of length k (k-mers) from the reference sequence. Trading off index size for potential sensitivity, such indices can either index all of the k-mers present in the underlying reference, or some uniform or intelligently-chosen sampling of k-mers. There are a large variety of k-mer-based aligners, including tools like the Subread aligner [90], SHRiMP2 [37], mrfast [3], and mrsfast [58]. To reduce the index size, one can choose to select specific k-mers based on a winnowing (or minimizer) scheme. This approach has been particularly common in tools designed for long-read sequence alignment like mashmap [71] and minimap2 [85].

Recently, a set of new indices for storing k-mers have been proposed based on graphs, specifically de Bruijn graphs (dBg). A de Bruijn graph is a graph over a set of distinct k-mers where each edge connects two neighboring k-mers that appear consequently in a reference sequence and therefore, overlap on "$k-1$" bases. Kallisto [23], deBGA [94], BGreat [91], BrownieAligner [61], and Pufferfish [7] are

some tools which use an index constructed over the de Bruijn graph built from the reference sequences. Cortex [67], Vari [111], rainbowfish [5], and mantis [123] are also tools that use a colored compacted de Bruijn graph for building their index over a set of raw experiments. All these approaches cover a wide range of the possible design space, and different design decisions yield different performance tradeoffs.

Generally, the fastest aligners (like STAR) have very large memory requirements for indexing, and make some sacrifices in sensitivity to obtain their speed. On the other hand, the most sensitive aligners (like Bowtie2) have very moderate memory requirements, but obtain their sensitivity at the cost of very high runtime. Maintaining the balance between time and memory is especially more critical while aligning to a large set of references, like a large collection of microbial and viral genomes which may be used as an index in microbiome or metagenomic studies. As both the collection of reference genomes and the amount of sequencing data growth quickly, it is import for alignment tools to achieve a time-space balance without loosing sensitivity.

Based on the compact Pufferfish [7] index, we introduce a new aligner PuffAligner, that we believe strikes an interesting and useful balance in this design space. PuffAligner is designed to be a highly-sensitive alignment tool while, simultaneously, placing a premium on computational overhead. By using the colored compacted de Bruijn graph to factor out repeated sub-sequences in the reference, it is able to leverage the speed and cache friendliness of hash-table based aligners while still controlling the growth in the size of the index; especially in the context of redundant reference sequences. By carefully exploring the alignment challenges that arise in different

assays, including single-organism DNA-seq, RNA-seq alignment to the transcriptome, and metagenomic sequencing, we have engineered a versatile tool that strikes desirable balance between accuracy, memory requirements and speed. We compare PuffAligner to some other popular aligners and show how it navigates these different tradeoffs.

2 Method

PuffAligner is an aligner on top of the Pufferfish index. Pufferfish is a space-efficient and fast index for the colored compacted de Bruijn graph (ccdBg). A colored compacted de Bruijn graph is defined as a graph where its vertices are the results of compacting the nodes (k-mers) in every non-branching path of the de Bruijn graph into a single node. The nodes in the colored compacted de Bruijn graph are called "unitig"s. Now, each unitig which contains all the k-mers in a longest monochromatic non-branching path in the de Bruijn graph can be mapped to a list of <reference ID, position, orientation> tuples. The output of Pufferfish index for a query sequence of length k (k-mer) is a list of raw hits or exact matches indicating the positions where each k-mer shows up in the underlying de Bruijn graph. This output is retrieved with one level of indirection for first finding the unitig containing the k-mer and then listing all the associated tuples. In PuffAligner, starting from these raw hits, we end up reporting a base-to-base alignment for each query to the reference sequences throughout a number of steps. Then, each raw hit is extended until reaching the end of the unitig or a mismatch happens. The exact

matches to the unitigs, called uni-MEMs, are then projected to the positions on the references associated to that unitig. Then, on each reference, the chains of exact matches with the highest coverage are selected. In the case of paired-end reads, the chains of the left and right ends are paired with respect to their distance, orientation, etc. Finally, rather than fully aligning each query sequence to the anchored position on the reference, only the sub-sequences from the query that are not part of the uni-MEMs (exact matches) are aligned to the reference, we call this procedure the between-MEM alignment. Each of these steps are explained in details in the following sections.

2.1 Exact matching in the Pufferfish index

Pufferfish index provides PuffAligner an efficient method for looking up k-mers within a list of references. Therefore, for each k-mer, all the references it appears in (with positions and orientations of the k-mer on that reference) are discovered very rapidly. Specifically, the core component of the index consist of (1) a minimal perfect hash function (MPHF), (2) a unitig sequence vector, (3) a unitig-to-reference table, and (4) a vector storing the position associated with each k-mer in the unitig sequence vector. The unitig sequence vector contains all the unitigs in the ccdBg. The Pufferfish index admits efficient exact search for k-mers, as well as longer matches that are unique in both the query string and colored compacted de Bruijn graph. These matches, called uni-MEM, were originally defined in deBGA [94]. A uni-MEM is a Maximal Extended Match (MEM) between the query sequence and a unitig.

Using the combination of the MPHF and the position vector, a k-mer is mapped to a unitig in the unitig sequence vector. The k-mer is then extended to a uni-MEM. Each uni-MEM can appear in different references. A MEM is then defined as a uni-MEM combined with a specific tuples of <reference, position, orientation>. A uni-MEM extension is terminated upon meeting any of these three conditions: (1) reaching a mismatch between the query and reference sequence (as the result of sequencing error or genomic variation), (2) reaching the end of the query, or (3) reaching the end of the unitig.

uni-MEM collection: The first step in read alignment is to collect exact matches shared between the query (single end or paired end reads) and the reference. In PuffAligner, this is accomplished by collecting the set of uni-MEMs that co-occur between the query and reference. PuffAligner starts processing the read from the left-end and looks up each k-mer that is encountered until a match to the index is found. Once a match is discovered, it is extended in both directions until a mismatch is encountered, or the end of the query or the unitig is reached. This process results in a uni-MEM match shared between the query and reference. If the uni-MEM extension is not terminated as a result of reaching the end of the query, then,the same procedure is repeated for the next k-mer on the read. This process continues until either the uni-MEM extension terminates because the end of the query is reached, or the last k-mer of the query is searched in the index. Here, we recall an important property of uni-MEM extension that is different from e.g. MEM extension or maximum mappable prefix (MMP) extension [40]; due to the definition

of the ccdBg, it is guaranteed that any k-mer appearing within a uni-MEM cannot appear in any other contig in the ccdBg. Thus, extending k-mers to maximal uni-MEMs is, in some sense, safe with respect to greedy extension, as such extension will never cause missing a k-mer that would lead to another distinct uni-MEM shared between the query and reference.

Filtering highly-repetitive uni-MEMs: In order to avoid expending a lot of computation on performing the subsequent steps on parts of the read mapping to highly-repeated regions of the reference, any uni-MEM that appears in more than a user-defined number of times in the reference is discarded. In this manuscript, we use the threshold of 1000. This filter has a strong impact on the performance, since, even if one k-mer from the read maps to a highly-repetitive region of the reference, the following expensive steps of the alignment procedure should be performed for every mapping position of the uni-MEM to find the right alignment for the read, while the less repetitive uni-MEMs also map to the true origin of the read on the reference too. The drawback of this filter is that for a very small fraction of the reads which are truly originating from a highly-repetitive region, all of the matched uni-MEMs will be filtered out and no k-mer hit remains for aligning the read. However, we find that in the case of aligning paired-end reads, usually one end of the read maps to a non-repetitive region, then, the alignment of the other end can be recovered using orphan recovery (explained in Section 2.4). Furthermore, using the option *–allowHighMultiMappers*, mitigates the effect of this filter by a very slight impact on the performance.

uni-MEM compaction: For paired-end reads, PuffAligner aligns each end the read pairs individually. For each end, all the uni-MEMs are sorted on the basis of their positions on the reference. Consecutive uni-MEMs with no gap (both on the reference and the read) are merged into larger MEMs. The compactable uni-MEMs are resulted from terminating the extension process due to reaching the end of a unitig, therefore, jumping to a new uni-MEM starting from the first base of another unitig for matching the rest of the query. Such consecutive uni-MEMs can be safely compacted to form longer MEMs that will be used later in the MEM chaining algorithm. After the compaction of uni-MEMs, there is a list of MEMs which are shared sequences between the query and a set of reference positions, that are sorted based on the reference positions.

2.2 Finding promising MEM chains

As shown in figure 5, having all the MEMs (maximal perfect matches) from a read to each target reference, the goal of this step is to find promising chains of MEMs that cover the most unique bases in the read and can potentially lead to a high quality alignment. To do so, we adopt the dynamic programming approach used in minimap2 [85] for finding co-linear chains of MEMs that are likely candidates to support high-scoring read alignments. As mentioned in minimap2, all the MEMs from a read r to the reference t, are sorted by the ending position of the MEMs on the reference. Then, this algorithm computes a coverage score for each set of MEMs based on the number of unique covered bases in the read, the coverage score

is also penalized by the length of the gaps, both in the read and reference sequence, between each two consecutive MEMs. Then, the set of the chains which yields the highest coverage from the read r to the reference t are selected through a dynamic programming approach.

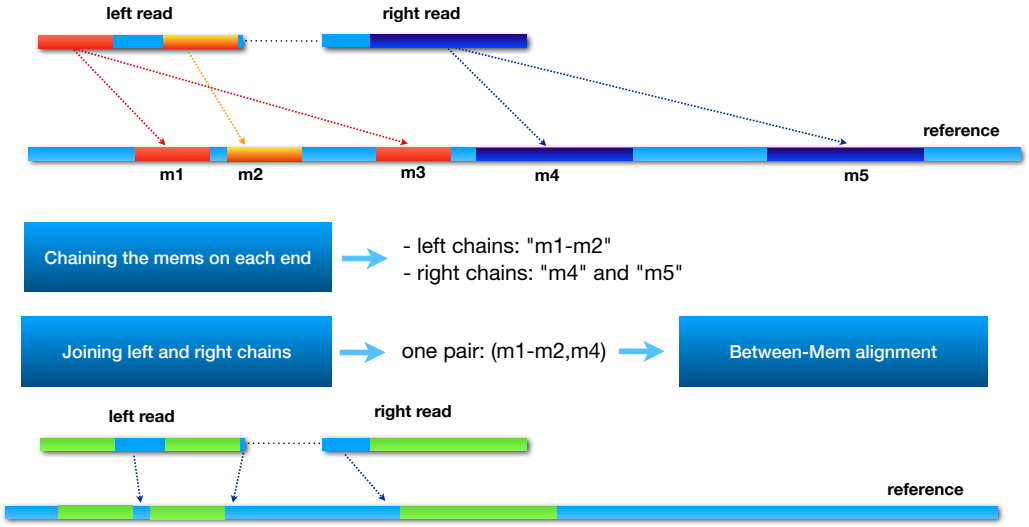


Figure 5: This figure shows the main steps of chaining and between-MEM alignment in the PuffAligner procedure via an example. In this example, m1, m2 and m3 are the projected MEMs from the left end of the read to the reference and m4 and m5 are the projected MEMs from the right end of the read. In the first step, the chaining algorithm chooses the best chain of MEMs that provide the highest coverage score for each end of the read, that is the m1-m2 chain for the left end and two single MEM chain for the right end. Then, the selected chains from each end are joined together to find the concordant pairs of chains, that is the (m1-m2, m4) pair for this read as m5 is too far from m1-m2. Then, the chain from each end will go through to the next step, between-MEM alignment. For the green areas (MEMs) no alignment is recalculated as they are exact matches. Only the un-matched blue parts of the chains (those nucleotides not occurring within a MEM) are aligned using a modified version of KSW2.

In puffaligner, if the distance between two MEMs, m_1 and m_2, on the read and the reference is d_r and d_t respectively, these two MEMs should not be chained together if $|d_r - d_t| > C$, where C is the maximum allowed splice gap. So, the penalization term, the β value in [85], in the coverage score computation is modified

accordingly to prevent pairing of such MEMs.

Also, unlike what is done in minimap2 [85], rather than considering together the MEMs that are discovered on both ends of a paired-end read, we consider the chaining and chain filtering for each end of the read separately. This is done in order to make it easier to enforce the orientation consistency of the individual chains. Specifically, the chaining algorithm that is presented in [85] introduces a transition in the recursion that can be used to switch between the MEMs that are part of one read and those that are part of the other. However, such switching makes it difficult to enforce the orientation consistency of the chains that are being built for each end of the read. A solution one can propose is to add another dimension to the dynamic programming table, encoding if one has switched from the MEMs of one read end to the other, the recurrence can be modified to allow only one switch from the one read end to the other, and to retain orientation consistency. However, we found that, in practice, simply chaining the read ends separately led to better performance.

Finally, we also adopt the heuristic proposed by [85] when calculating the highest scoring chains. That is, when a MEM is added to the end of an existing chain, it is unlikely that a higher score for a chain containing this MEM will be obtained by adding it to a preceding chain. Thus, we consider only a small fixed number of rounds (by default 2) of preceding chains once we have found the first chain to which we can add the current MEM.

The chaining algorithm described above finds the best chains of MEMs shared between the read r and the reference t in orientation o. A chain is accepted if its coverage score is greater than a configurable fraction, which we call the *consensus-*

Fraction, times the maximum coverage score found for the read r to *any* reference. Throughout all the experiments in this manuscript the *consensusFraction* is set to 0.65. If a chain passes the consensus fraction threshold, we call it a *valid* chain. Additionally, rather than keeping all valid chains, we also filter highly-suboptimal chains with respect to the highest scoring chain per reference. All valid chains shared between r and t are sorted by their coverage scores, and chains having scores within 10 percent of the highest scoring chain for reference t are selected as potential mappings of the read r to the reference t. While these filters are essential for improving the throughput of the algorithm in finding the right alignment, they are carefully selected to have very little effect on the sensitivity of PuffAligner. For all the experiments in this manuscript, the same default settings of these parameters are used if not mentioned otherwise.

2.3 Computing base-to-base alignments between MEMs

After finding the high-scoring MEM chains for each reference sequence, a base-to-base alignment of the read to each of the candidate reference sequences is computed. Each selected chain implies a position on the reference sequence where the read might exhibit a high quality alignment. Thus, we can attempt to compute an optimal alignment of the read to the reference at this implied position, potentially allowing a small bit of padding on each side of the read. This approach utilizes the positional information provided by the MEM chains. However, the starting position of the alignments is not the only piece of information embedded in the chains. Rather

each chain of MEMs consists of sub-sequences of the read (of size at least k) which match exactly to the reference. While the optimal alignment of the read to the reference at the position being considered is not *guaranteed* to contain these exact matches as alignments of the corresponding substrings, this is almost always the case.

In PuffAligner, we aim to exploit the information from the long matches to accelerate the computation of the alignments. In fact, since only chains with the relatively high coverage score are selected, a large portion of the read sequences are typically already matched to the positions in the reference with which they will be matched in the final optimal alignment. For instance, in Fig. 5, for the final chains selected on the reference sequence, it is already known for the light blue, dark blue and green sub-sequences on the left end of the read precisely where they should align to the reference. Likewise for the yellow and purple sub-sequences on the right read. The unmapped regions of the reads are either bordered by the exact matches on both reference and read, or they occur at the either ends of the read sequence. PuffAligner skips aligning the whole read sequence by considering the exact matches of the MEMs to be part of the alignment solution. As a result, it is only required to compute the alignment of the small unmapped regions, which reduces the computation burden of the alignments.

When applying such an approach, two different types of alignment problems are introduced, which we call bounded sub-sequence alignment and ending sub-sequences. For bounded sub-sequence alignment, we need to *globally* align some interval i_r of the read to an interval i_t of the reference. If i_r and i_t are of different

lengths, the alignment solution will necessarily include insertions or deletions. If i_r and i_t are of the same length, then the optimal global alignment between them may or may not include indels. For each such bounded sub-sequence alignment, we determine the optimal alignment of i_r to i_t by computing a global pair-wise alignment between the intervals, and stitching the resulting alignment together with the exact matches that bound these regions.

Gaps at the beginning or the end of the read are symmetric cases, and so we describe, without loss of generality, the case where there is an unaligned interval of the read after the last MEM shared between the read and the reference. In this case, we need to solve the ending sub-sequence alignment problem. Here, the unaligned interval of the read consists of the substring spanning from the last nucleotide of the terminal MEM in the chain, up through the last nucleotide of the read. There is not a clearly-defined interval on the reference sequence. While the left end of the relevant reference interval is defined by the last reference nucleotide that is part of the bounding MEM, the right end of the reference interval should be determined by actually solving an extension or "end-free" alignment problem. We address this by performing extension alignment of the unaligned interval of the read to an interval of the reference that begins on the reference at the end of the terminal MEM, and extends for the length of the unaligned query interval plus the length of some problem-dependent buffer (which is determined by the maximum length different between the read and reference intervals that would still admit an alignment within the acceptable score threshold).

An example of both of these cases is displayed in Figure 5. Specifically, align-

ment of the read could be obtained by only solving two smaller alignment problems; one is the ending sub-sequence alignment of the unmapped region after the green MEM on the left read and the other is the bounded sub-sequence alignment of region on the right read bordered by the yellow and purple MEMs.

PuffAligner uses KSW2 [85, 154] for computing the alignments of the gaps between the MEMs. KSW2 exposes a number of alignment modes such as global and extension alignments. For aligning the bounded regions, KSW2 alignment in the global mode is performed, and for the gaps at the beginning or end of reads, PuffAligner uses the extension mode to find the best possible alignment of that region. PuffAligner, by default, uses a match score of 2 and mismatch penalty of 4. For indels, PuffAligner uses an affine gap scoring schema with gap open penalty of 5 and gap extension penalty of 3. In PuffAligner, after computing the alignment score for each read, only the alignments with a score higher than τ times the maximum possible score for the read are reported. The value of τ is controlled by the option *–minScoreFraction*, which is set to 0.65 by default.

2.3.1 Enhancing alignment computation

Although, by only aligning the read's sub-sequences that are not included in the MEMs, the size of alignment problems being solved in PuffAligner are often much shorter than the length of the read, we also incorporate a number of other techniques to improve the performance of the alignment calculation even further. We describe the most important of these below:

- **Skipping alignment calculation by recognizing perfect chains and alignment caching:** It is possible to avoid the alignment computation completely in a considerable number of cases. In fact, it has been explored in previous work [141] that the alignment calculation step can be completely skipped if the set of exact matches for each chain covers the whole read. PuffAligner skips alignment for cases where the coverage score of chains of MEMs is the length of the read, and assigns a total matched CIGAR string for that alignment. Alignment computation of a read might be also skipped if the same alignment problem has been already detected and computed for this read. For example, in the case of RNA seq data, reads often map to the same exons on different transcripts. In such cases, each alignment solution for a read is stored in a cache (a hash table) so that if the same alignment problem is detected, the solution can be directly retrieved from the cache, and no further computation is required (see supplementary Table 7).

- **Early stopping of the alignment computation when a valid score cannot be achieved:** While care is taken to produce only high-scoring chains between the read and reference, it is nonetheless the case that the majority of the chains do not lead to an alignment of acceptable quality. Since the minimum acceptable alignment score is immediately known based on τ and the length of the read, the base-to-base alignment calculation can be terminated at any point where it becomes apparent that the minimum required alignment score cannot be obtained. This approach can be applied both during the

KSW2 alignment calculation, and also after the alignment calculation of each gap is completed. During this procedure, for each base at position i, starting from 1 on the read of length n, if the best alignment score p up to the i-th position is s_i, we can calculate the maximum possible alignment score, s_{max}, that might be achieved starting at this location given the current alignment score by:

$$s_{max} = s_i + MS * (n - s_i), \tag{3.1}$$

where MS is the score assigned to each match. If MS is smaller than minimum required score for accepting the alignment, the alignment calculation can be immediately terminated, since it is already known that this anchor is not going to yield a valid alignment for this read.

- **Maximum allowed gap length:** KSW2 is able to perform banded alignment to make alignment calculation more efficient. By calculating the maximum number of gaps (insertions or deletions) allowed in each sub-alignment probem, in a way that the total alignment score does not drop below the accepted threshold, we utilize the banded alignment in KSW2 without losing any sensitivity.

2.4 Joining mappings for read ends and orphan recovery

Finally, once alignments have been computed for the individual ends of a read, they must be paired together to produce valid alignments for the entire fragment. At this point in the process, on each reference sequence, there are a number of

locations where the left end of each read or the right end of each read or both map. For the purpose of determining which mappings will be reported as a valid pair, the mappings are joined together only if they occur on opposite strands of the reference, and if they are within a maximum allowed fragment length. There are two different types of paired-end alignments that can be reported by PuffAligner; concordant and discordant. If PuffAligner is disallowed from reporting discordant alignments, then the mapping orientation of the left and right end should agree with the library preparation protocols of the reads. PuffAligner first tries to find concordant mapping pairs on a reference sequence, and if no concordant mapping is discovered and the tool is being run in a mode where discordant mappings are allowed, then PuffAligner reports pairs that map discordantly. Here, discordant pairs may be pairs that do not, for example, obey the requirement of originating from opposite strands. While this is not expected to happen frequently, it may occur if there has been an inversion in the sequenced genome with respect to the reference.

Orphan recovery: If there is no valid paired-end alignment for a fragment (either concordant or discordant, if the latter is allowed), then PuffAligner will attempt to perform orphan recovery. The term "orphan" refers to one end of paired-end read that is confidently aligned to some genomic position, but for which the other read end is not aligned nearby (and paired). To perform orphan recovery, PuffAligner examines the reference sequence downstream of the mapped read (or upstream if the mapped read is aligned to the reverse complement strand) and directly performs

dynamic programming to look for a valid mapping of the unmapped read end. For this purpose, we use the "fitting" alignment functionality of edlib [151] to perform a simple 0/1 edit-distance based alignment that will subsequently be re-scored by KSW2. Finally, if, after attempting orphan recovery, there is still no valid paired-end mapping for the fragment, then orphan alignments are reported by PuffAligner (unless the "`--noOrphans`" flag is passed).

3 Evaluation

For measuring the performance of PuffAligner and comparing it to other aligners, we have designed a series of experiments using both simulated and experimental data from different sequencing assays. We compare PuffAligner with Bowtie2 [78], STAR [40] and deBGA [94]. Bowtie2 is a popular, sensitive and accurate aligner with the benefit of having very modest memory requirements. STAR requires a much larger amount of memory, but is much faster than Bowtie2 and can also perform "spliced alignment" against a reference (which PuffAligner, Bowtie2, and deBGA currently do not allow). deBGA, is most-related tool to PuffAligner conceptually, as it is an aligner with a colored compacted de Bruijn graph-based index that is focused on exploiting redundancy in the reference sequence.

We use different metrics to assess both the performance and accuracy of each method on a variety of types of sequencing samples. These experiments are designed to cover a variety of different use-cases for an aligner, spanning the gamut from situations where most alignments are expected to be unique (DNA-seq), to situations

where each fragment is expected to align to many loci with similar quality (RNA-seq and metagenomic sequencing), and spanning the range of index sizes from small transcriptomes to large collections of genomes.

First, we show PuffAligner's exhibits similar accuracy for aligning DNA-seq reads to Bowtie2, but it is considerably faster. In the case of experimental reads, since the true origin of the read is unknown, we use measures such as mapping rate and concordance of alignments to compare the methods. Furthermore, we evaluate the accuracy of aligners by aligning simulated DNA-seq reads that include variation (single-nucleotide variants and small indels with respect to the reference). For aligning RNA-seq reads, we compare the impact of alignments produced by each aligner on downstream analysis such as abundance estimatation. Finally, we show PuffAligner is very efficient for aligning metagenomic samples where there is a high degree of shared sequence among the reference genomes being indexed. We also illustrate that using alignments produced by PuffAligner yields the highest accuracy for abundance estimation of metagenomic samples.

3.1 Configurations of aligners in the experiments

The performance of each tool is impacted by the different alignment scoring schemes they use, e.g. different penalties for mismatches, and indels. To enable a fair comparison, we attempted to configure the tools so as to minimize divergences that simply result from differences in the scoring schemes. For the experiments in this paper, we use Bowtie2 in a near-default configuration (though ignoring quality

values), and attempt to configure the other tools, as best as possible, to operate in a similar manner.

The deBGA scoring scheme is not configurable, so we use this aligner in the default mode (unfortunately, the inability to disable local alignment and forcing just computation of end-to-end alignments in deBGA makes certain comparisons particularly difficult). For PuffAligner we use a scheme as close to Bowtie2 as possible. The maximum possible score for a valid alignment in Bowtie2 is 0 (in end-to-end mode) and each mismatch or gap subtracts from this score. Bowtie2 uses an affine gap penalty scoring scheme, where opening and extending a gap (insertion or deletion) have a cost of 5 and 3 respectively. For DNA-seq reads, we configure STAR to allow as many mismatches as Bowtie2 and PuffAligner by setting the options "`–outFilterMismatchNoverReadLmax 0.12`" and "`–outFilterMismatchNmax 1000`". Also, we use the option "`–alignIntronMax 1`" in STAR to perform non-spliced alignments while aligning genomic reads. For RNA-seq reads, STAR has a set of parameters which we change in our result evaluations, and which are detailed below in the relevant sections.

In Bowtie2 we also use the option `–gbar 1` to allow gaps anywhere on the read except within the first nucleotide (as the other tools have no constraints on where indels may occur). Furthermore, for consistency, we also run Bowtie2 with the option "`–ignore-quals`", since the other tools do not utilize base qualities when computing alignment scores.

As explained in Section 2.1, for the sake of performance, highly repeated anchors (more than a user-defined limit) will be discarded before the alignment phase.

This threshold is by default equal to 1000 in PuffAligner. We set the threshold to the same value for STAR and deBGA using options `–outFilterMultimapNmax 1000` and `-n 1000` respectively. There is no such option exposed directly in Bowtie2.

Since PuffAligner finds end-to-end alignments for the reads, we are also running other tools in end-to-end mode, which is the default alignment mode in Bowtie2 as well. In STAR we enable this mode using the option `–alignEndsType EndToEnd`. In the case of deBGA, although the documentation suggests it is *not* supposed to find local alignments by default, the output SAM file contains many reads with relatively long soft clipped ends, so if a read is not aligned end-to-end, deBGA reports the local alignment for that. We were not able to find any option to force deBGA to perform end-to-end alignments for all reads, and so we have compared it in the configuration in which we were able to run it.

For aligning DNA-seq samples, each aligner is configured to report a single alignment, which is the primary alignment, for each read. Bowtie2 outputs one alignment per read by default. To replicate this in the other tools, we use the option `–outSAMmultNmax 1` in STAR, `-o 1 -x 1` in deBGA, and `–primaryAlignment` in PuffAligner.

3.2 Alignment of whole genome sequencing reads

First, we evaluate the performance of PuffAligner with a whole genome sequencing (WGS) sample from the 1000 Genomes project [32].We downloaded the `ERR013103` reads from sample `HG00190`, which is a low-coverage sample from a Finnish male,

sequenced in Finland.*. There are 18, 297, 585 paired-end reads, each of length 108 nucleotides in this sample. Using fastp [27], we remove low quality ends and adapter sequences from these reads. After trimming, there are 15, 404, 412 reads remaining in the sample. Indices for each of the tools are built over all DNA chromosomes of the latest release of the human genome (v33) by gencode† [48].

In this experiment, all aligners are configured report only concordant alignments, i.e., only pairs of alignments that are cocordant and within the "maximum fragment length" shall be reported. The maximum fragment length in all aligners is set to 1000, using the option `–alignMatesGapMax 1000` in STAR, `–maxins 1000` in Bowtie2 and `-u 1000 -f 0` in deBGA. The default value for the maximum fragment length in PuffAligner is set to 1000, the user can cofigure this value by using the flag `–maxFragmentLength`. This concordance requirements also prevents Bowtie2, PuffAligner, and STAR from aligning both ends of a paired end read to the same strand.

The alignment rate, run-time memory usage and running time for all the aligners are presented in 3. The reason that deBGA has the highest mapping rate in 3 compared to other tools is that it is local alignments for the reads that are not alignable end-to-end under the scoring parameters for the other tools. Bowtie2 and PuffAligner are both able to find end-to-end alignments for about $\sim 95\%$ of the reads. STAR and PuffAligner are the fastest tools, with STAR being somewhat faster than PuffAligner. On the other hand, PuffAligner is able to align more reads

*https://www.internationalgenome.org/data-portal/sample/HG00190
†https://www.gencodegenes.org/human/release_33.html

than STAR, while requiring less than half as much memory. The memory usage of Bowtie2 is the smallest, since Bowtie2's index does not contain a hash table. However, this comes at the cost of having the longest running time compared to other methods. Overall, PuffAligner benefits from the fast query of hash based indices while its run-time memory usage, which is mostly dominated by the size of the index, is significantly smaller than other hash based aligners. Although deBGA's index is based on the de Bruijn graphs, similar to the Pufferfish index, the particular encoding for it is not as space-efficient as that of Pufferfish.

aligner	mapping-rate(%)	time (mm:ss)	memory (GB)
PuffAligner	95.58	6:14	13.09
deBGA	99.75	10:46	41.04
STAR	93.88	4:29	30.36
Bowtie2	95.44	16:15	3.50

Table 3: The performance of different tools for aligning experimental DNA-seq reads. The time reports are benchmarked after warming up the system cache so that the influence of index loading time is mitigated.

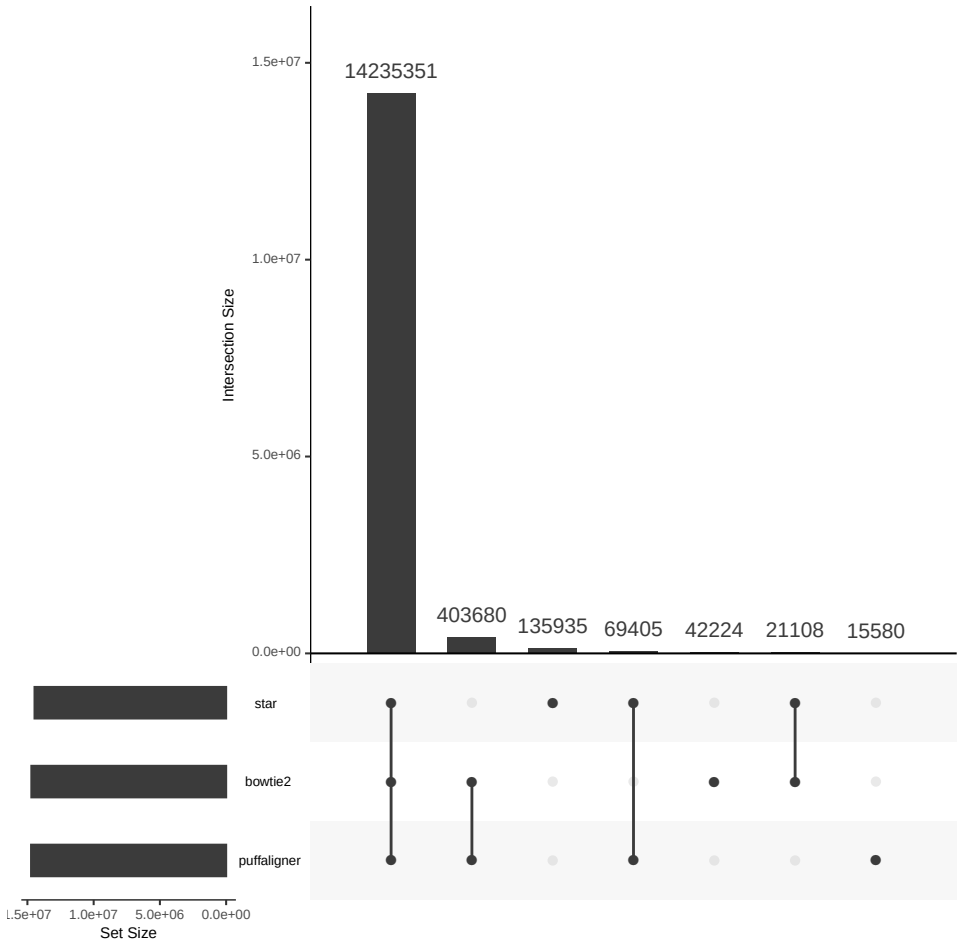


Figure 6: Upset plot showing the agreement of the alignments found by different tools

To look more closely how the mappings between the tools differ, we investigate the agreement of the reads which are mapped by each tool and visualize the results in an upset plot in Fig. 6 using the UpsetR library [34]. We are only comparing the three methods which perform end-to-end alignment in this plot, since outliers from the local alignments computed by deBGA would otherwise dominate the plot. The first bar shows that the majority of the reads are mapped by all three tools.The next largest set represents the reads which are only mapped by Bowtie2 and PuffAligner. All the other sets are much smaller compared to the first two sets. This fact illustrates that the highest agreement in the aligners is between Bowtie2

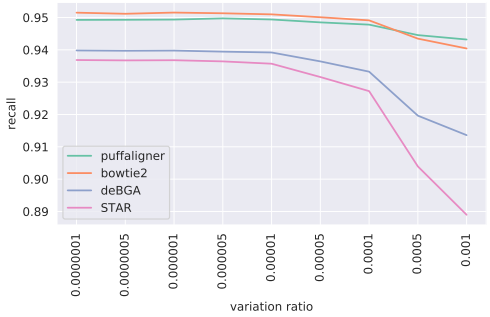
and PuffAligner. Exploring a series of individual reads from the smaller sets in the upset plot, suggests that some of these differences happen as a result of small differences in the scoring configuration, while some result from different search hueristics adopted by the different tools. Supplementary Fig. 11 shows the coherence between the alignments reported by the tools by also including the exact location to which the reads are aligned in the reference.

3.3 Alignment of simulated DNA-seq reads in the presence of variation

To further investigate the accuracy of the aligners, we used simulated DNA-seq reads.One of the main differences between simulated reads and experimental reads is that simulated reads are often generated from the same reference sequences to which they are aligned, with the only differences being due to (simulated) sequencing error. While (simulated) sequencing error prevents most reads from being exact substrings of the reference, it actually does not tend to complicate alignment too much. On the other hand, while dealing with experimental data, the genome of the individual from which the sample is sequenced might include different types of variations with respect to the reference genome to which we are aligning [152]. Therefore, it is desirable to introduce variations in the simulated samples, and to measure the robustness and performance of the different aligners in the presence of the variation. Mason [64] is able to introduce different kinds of variations to the reference genome, such as SNVs, small gaps, and also structural variants (SV)

such as large indels, inversions, translocations and duplications. We use Mason to simulate 9 DNA-seq samples with different variation rates ranging from $1e-7$ to $1e-3$. Each sample includes $1M$ paired-end Illumina reads of 100bp length from chromosome 21 of the human genome, ensembl release 98*.

(a) The precision of the alignments reported by each aligner. True positives (TP) are the compatible reads that are aligned to the original location, and the FP set consists of both the compatible reads aligned to sub-optimal locations (alignments with larger edit distance than the alignment to the original location) and the non-compatible reads that are aligned with high (>25) edit distance.



(b) The ratio of the alignments in the true SAM file that are recovered by each aligner. The recall is the result of dividing the number of TP reads by the total number of compatible reads.

Figure 7: Comparing the accuracy of different aligners in the presence of different rates of variations in the reference genome

For this analysis, we do not restrict the aligners to only report concordant alignments, since the structural variations in the samples can lead to valid discordant alignments, such as those on the same strand or with inter-mate distances larger than the maximum fragment length. To be specific, we do not use the options which limit Bowtie2 and PuffAligner to report only concordant alignments, in addition, we use the option "`--dovetail`" in Bowtie2 to consider dovetail pairs as concordant pairs.

The alignments reported by deBGA already include discordant pairs and also

*ftp://ftp.ensembl.org/pub/release-98/fasta/homo_sapiens/dna/

orphan mappings. Furthermore, To remove any restrictions on the fragment length in the alignments reported by deBGA, we set the minimum and maximum insert size, respectively to 0 and the 50000, since setting a larger value resulted in the tool running into segmentation fault.

To allow dovetail pairs and also larger gaps between the pairs in STAR, we use the following options: "`–alignEndsProtrude 1000000 ConcordantPair`", "`–alignMatesGapMax 1000000`". By default there is not a specific option in STAR for allowing orphan alignment of paired end reads. Instead, we can increase the number of allowed mismatches to be as large as one end of the read by using the following options:

"`–outFilterMismatchNoverReadLmax 0.5`",

"`–outFilterMismatchNoverLmax 0.99`",

"`–outFilterScoreMinOverLread 0`",

"`–outFilterMatchNminOverLread 0`".

For each sample, Mason produces a SAM file which includes the alignment of the simulated reads to the original, non-variant version of the reference — the version which was used for building the aligner's indices in this experiment. Based on the alignments reported in the truth file, some reads did not have a valid alignment to the original reference. This was the result of a high rate of variations at some sequencing sites. We called the set of reads that, according to the truth SAM file, were aligned to the original reference as compatible reads.

We compared the performance of aligners based upon how well they are able to align the compatible reads. We computed the precision and recall of the alignments

reported for these reads as follows. True positives are considered the reads that are mapped by the aligner to the same location stated by the truth file. Then, recall is computed by dividing the number of true positives by the number of all compatible reads. Furthermore, we considered an alignment as a false positive in two different cases. First, an alignment was considered discordant if the reported alignment had a large edit distance (larger than 25) for the non-compatible reads. Second, in the case that an aligner reported an alignment to a location other than the one in the truth file, it was considered as a false positive if the edit distance of the reported alignment is greater than the edit distance of the true alignment. Having defined the set of TP and FP for the alignments, and also having considered the set of all compatible reads as the set we are trying to recover, we computed precision and recall for the set of alignments reported by each aligner.

Figure 7 shows the precision and recall of the aligners for different samples. According to Fig. 7, for lower variation ratios up until $10e-5$, most of the tools are able to make accurate alignment calls with a high specificity. As the variation ratio introduced in the sample is increased, all the tools start to have lower precision and recall. deBGA and STAR perform worse in higher variation samples, as they fail to recover the true alignment for more reads, while Bowtie2 and PuffAligner are able to align most of the reads to their true location on the original reference.

These results show that PuffAligner' accuracy is stable in the face of variation which makes the tool suitable for datasets that are known to have substantial variation, such as when aligning reads to microbial genomes where the specific sequenced strain may not be represented in the reference set.

3.4 Quantification of RNA-seq reads

Mapping sequencing reads to target transcriptomes is the initial step in many pipelines for reference-based transcript abundance estimation. While lightweight mapping approaches [23, 127] greatly speed-up abundance estimation by, in part, eliding the computation of full alignment between reads and transcripts, there is evidence that alignments still yield the most accurate abundance estimates by providing increased sensitivity and avoiding spurious mappings [141, 152]. Thus, the continued development of efficient methods for producing accurate transcriptome alignments of RNA-seq reads remains a topic of interest. In this section, we compare the effect of alignments produced by each tool on the accuracy of RNA-seq abundance estimation.

We generated 9,968,245 paired-end RNA-seq reads using the polyester [49] read simulator. The reads are generated by the `simulate experiment countmat` module in polyester. The input count matrix is calculated based on the estimates from the Bowtie2-Salmon pipeline on the sample `SRR1085674` (where reads are first aligned with Bowtie2 and then the alignments are quantified using Salmon). This sample is a collection of paired-end RNA-seq reads sequenced from human transcriptome using an Illumina HiSeq [98]. The human transcriptome from gencode release (33) is used to build all the aligners' indices. Also, for building STAR's index in the genome mode, the human genome and the comprehensive gene annotation (main annotation file) is obtained from the same release of gencode.

As the reads in this experiment are RNA-seq reads sequenced from the hu-

aligner	spearman	MARD	time (mm:ss)	memory (GB)
PuffAligner	0.92	0.05	1:17	2.54
deBGA	N/A	N/A	5:19	9.96
STAR- transcriptome	0.92	0.05	1:57	8.73
STAR- genome	0.90	0.06	3:30	32.57
Bowtie2	0.92	0.05	32:59	1.15

Table 4: Abundance estimation of simulated RNA-seq reads, computed by Salmon, using different tools' alignment outputs. The time and memory are only for the alignment step of each tool and the time for abundance estimation by Salmon is not considered.

man transcriptome, it is important to account for multi-mapping, as often, a read might map to multiple transcripts which share the same exon or exon junction. This property makes the direct evaluation of performance at the level of alignments difficult. Therefore, a typical approach in evaluating the accuracy of the transcriptomic alignments is to assess the accuracy of downstream analysis such as abundance estimations by computing the correlation and relative differences of the estimates with the true abundance of the transcripts. To compare the accuracy of each tool we give the alignments produced by each aligner, which are in the SAM format, as input to Salmon to estimate the transcript expressions.

PuffAligner, by default, outputs up to 200 alignments with an alignment score greater than 0.65 times the best alignment score, i.e., the alignment for the read in the case that all bases are perfectly matched to the reference. To enable the multi-mapping to take into account the characteristics of alignment to the transcriptome, Bowtie2 is run with the option `-k 200` which lets the tool output up to 200 alignments per read. The value of 200 is adopted from the suggested parameters for running RSEM [82] with Bowtie2 alignments. We note that running Bowtie2 with this option makes the tool considerably slower than the default mode, as many

more alignments will be computed and output to the SAM file under this configuration. For both Bowtie2 and PuffAligner, and also for STAR by default, orphan and discordant mappings are not allowed.

We ran STAR with the `'ENCODE'` options, which are recommended in the STAR manual for RNA-seq reads. STAR is also run in two different modes, one is by building the STAR index on human genome, while it is also provided a GTF file for gene annotation. In this mode, STAR performs spliced alignment to the genome, then projects the alignments onto transcriptomic coordinates. The other mode is building the STAR index on the human transcriptome directly, which allows STAR to align the RNA-seq reads directly to the transcripts in an unspliced manner. We chose to run STAR in the transcriptomic mode as well, since we find that it yields higher accuracy, though this increases the running time of STAR.

The deBGA index is built on the transcriptome, as are the Bowtie2 and PuffAligner indices, since these tools do not support spliced read alignment. deBGA is run in the with options `-o 200 -x 200`, which nominally has the same effect as `-k 200` in Bowtie2, according to the documentation of deBGA.

Accuracy of abundance estimation by Salmon, when provided the SAM output generated by each aligner, is displayed in Table 4. The timing and memory benchmarks provided in this table is only for the alignment step. Alignments produced by PuffAligner, Bowtie2 and STAR in the transcriptomic mode produce the best abundance estimates. deBGA's output alignments are not suitable for any abundance estimation as many reads are aligned only to the same strand which are later filtered during the abundance estimation by Salmon, so we could not provide a meaning-

ful correlations for abundance estimation using deBGA's alignments. Aligning the reads by STAR to genome and then projecting to transcriptomic coordinates does not generate as high correlation as directly aligning the reads to the transcriptome by STAR. However, we note that, as described by Srivastava et al. [152], there are numerous reasons to consider alignment to the entire genome that are not necessarily reflected in simulated experiments. While the memory usage by PuffAligner is only 2 fold larger than memory used by Bowtie2, it computes the alignments much more quickly.

According to the results in Table 4 PuffAligner is the fastest aligner in these benchmarks, and the accuracy as high as Bowtie2 and STAR for aligning RNA-seq reads. Here, PuffAligner leads to the most accurate abundance estimates, while being 30 times faster than Bowtie2. Moreover, The memory usage is much less than other fast aligners such as STAR.

3.5 Alignment to a collection of microorganisms — simulated short reads

To demonstrate the performance and accuracy of PuffAligner for metagenomic samples, we designed two different experiments. One main property of metagenomic samples is the high similarity of the reference sequences against which one typically aligns, where a pair (or more) of references may be more than 90% identical. The first experiment we designed for this scenario, to specifically evaluate issues related to this challenge, we call the "single strain" experiment. Additionally, metagenomic

samples also have the property of containing reads from a variety of genomes, some of which are not even assembled yet – and hence unknown. This leads to the second experiment, which we call the "bulk" experiment, that compares the aligners in the presence of a high variety of `species` in the sample in addition to the high similarity of references.

For simplicity and uniformity, all the experiments have been run in the concordant mode for both PuffAligner and Bowtie2 (both of which support such an option), disallowing orphans and discordant alignments. All aligners are run in three different confiurations, allowing three specific maximum numbers of alignments per fragment; 1 (primary output with highest score, breaking ties randomly), 20, and 200. PuffAligner and STAR, as the only tools that support this option, also are run in the *bestStrata* mode. In this mode, the aligner outputs all *equally-best* alignments for a read with highest score without the limitation on number of reported alignments. This option is inspired by the similarly-named option in Bowtie1 [77]. However, unlike Bowtie1, PuffAligner and STAR only make a best-effort attempt to find the score of the best stratum alignments, and do not guarantee to find the best stratum (though the cases in which they fail to seem to be exceedingly rare). This option is especially useful in the metagenomic analyses, as we will report only the best-score alignments without having an arbitrary limitation on the number of allowed alignments. This allows proper handling of highly multi-mapping metagenomic reads. In other words, using this option, one can achieve a high sensitivity without the need to hurt specificity. The details of each experiment is explained in the following sections.

3.5.1 Single-strain Experiment

For this experiment, we download the viral database from NCBI, and choose three similar coronavirus genomes. This set includes one of the recently-uploaded samples from Wuhan [12, 162]. We select three very similar viral genomes to simulate reads from which are: NC_045512.2, NC_004718.3, and NC_014470.1. There are also a lot of literature discussing the similarity in sequence and behavior for these three species of coronavirus [157, 159, 167]. The first is the complete genome for severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1 known as Covid19 with length of $29,904$ bases. NC_004718.3 is the ID of SARS coronavirus complete genome (length: $29,752$) and finally, NC_014470.1 is a Bat coronavirus BM48-31/BGR/2008 complete genome (length: $29,277$).

We use Mason [64] to generate three simulated samples, each sample contains $500,000$ reads only from one of the three viral references we mentioned earlier. Then, reads were aligned back to the database of viral sequences using each of the four aligners. The results are shown in table 5 for covid19 and table 10 for the other two simulations.

As the results show, the alignments of all aligners, except for deBGA, are distributed only across the three references of interest out of all the reference sequences in the complete viral database. deBGA reports only a few alignments to a forth virus. The results show that as we allow more alignments to be reported, sensitivity increases for all the tools, while specificity decreases, meaning more alignments to the wrong reference are reported. However, the results do not change when allowing

more than 20 alignments, which means no more than 20 alignments ever pass the alignment score threshold for these reads in the viral database.

The results indicate that, when allowing more than one alignment to be reported for every read, Bowtie2 reports a lot of false positive alignments compared to other tools. These are alignments that are accepted within the alignment score threshold, but are to another target than the one the read originates from.

Interestingly, there is one read that all tools, except for PuffAligner miss. Inspecting this alignment reveals it is a valid alignment within the range of the acceptable scoring threshold, and it is unclear why it is not discovered by the other tools. Overall, most aligners perform well here, specifically PuffAligner shows a good balance in sensitivity and specificity. Furthermore, based on the result in figure 8, section 8a, PuffAligner has the best performance in terms of running time even when the number of allowed alignments per read increases.

BestStrata Mode As expected, the sensitivity of all tools improve when allowing more alignments to be reported, since there is a higher chance for all aligners to report the alignment to the true origin of the read. PuffAligner, however, achieves the perfect sensitivity even in the primary mode, when it reports one alignment per read. After increasing the number of reported alignments per read, it still reports all of the right alignments, while reporting some false positive alignments to other reference sequences, which hurts its specificity in this case.

This is a small test for multi-mapping cases, but in larger samples, allowing more alignments usually yields better sensitivity. To control the false positive rate,

PuffAligner supports the "best strata" option – also available to STAR, which allows only the alignments with the best calculated score to be reported (as a replacement for maximum allowed number of alignments). Using this option, PuffAligner can reach perfect specificity and sensitivity in this experiment 5. The same results are achieved for the other two simulated single-strain samples shown in the supplementary table 10. We further demonstrate the postitive impact of this option on the alignment of bulk metagenomic samples in the next section.

Alignment Mode	Tool	NC_045512.2 (Covid19)	NC_004718.3 (SARS Coronavirus)	NC_014470.1 (Bat Coronavirus)	Other Assembled References
Primary	PuffAligner	500,000	0	0	0
	Bowtie2	499,981	18	0	0
	STAR	499,999	0	0	0
	deBGA	499,991	0	0	9
Up to 20	PuffAligner	500,000	134	46	0
	Bowtie2	499,999	21,461	2,311	0
	STAR	499,999	0	0	0
	deBGA	499,991	0	0	9
Up to 200	PuffAligner	500,000	134	46	0
	Bowtie2	499,999	21,461	2,311	0
	STAR	499,999	0	0	0
	deBGA	499,991	0	0	9
Best strata	PuffAligner	500,000	0	0	0
	STAR	499,999	0	0	0

Table 5: Alignment distribution for 500000 simulated reads from reference sequence NC_045512.2 (known as covid19). The best specificity is achieved by PuffAligner in `bestStrata` mode (as well as the primary mode). In this simulated sample, many alignments are not ambiguous, resulting in the good performance observed when using only primary alignments. However, typically in metagenomic analysis, many equally-good alignments exist, and selecting only one is equivalent to making a random choice.

3.5.2 Bulk Experiment

We chose a random set of 4000 complete bacterial genomes downloaded from the NCBI microbial database and constructed the indices of PuffAligner, Bowtie2, STAR, and deBGA on the selected genomes. Supplementary Table 8 shows the

time and memory required for constructing each of the indices, in addition to the size of the final index on disk. Overall, PuffAligner and Bowtie2 show a pretty similar trend in time and memory requirements while STAR and deBGA require an order of magnitude more memory.

For simulating a bulk metagenomic sample, we generated a list of simulated whole genome sequencing (WGS) reads through the following steps:

- Select a real metagenomic WGS read sample
- Align the reads of the chosen real experiment to the 4000 genomes using Bowtie2, limiting Bowtie2 to output one alignment per read.
- Choose all the references with count greater than C from the quantification results. This defines the read distribution profile that we will use to simulate data.
- For each of the expressed references, use Mason [64], a whole genome sequence simulator, to simulate $100bp$ paired-end reads with counts proportional to the reported abundance estimates so that total number of reads is greater than a specified value n. In this step we ran Mason with default options.
- Mix and shuffle all of the simulated reads from each reference into one sample which is used as the mock metagenomic sample.

We selected three Illumina WGS samples that are publicly available on NCBI. A soil experiment with accession ID `SRR10948222` from a project for finding sub-biocrust soil microbial communities in the Mojave Desert. The sample has $\sim 27M$

paired-end reads, containing a mixture of genomes from various genera and families. However, less than $200k$ of the reads in the sample were aligned to the strains present in our database, leading the selection of 98 species from a variety of genera. We scaled the read counts in the simulation to $\sim 50M$ reads. The other two selected samples are `SRR11283975` and `SRR11496426` the details of which are explained in supplementary 9. In this section we only report the performance of the tools on the first sample. The analysis results for the other samples (which shows similar relative accuracy and performance for different tools) are provided in 11.

The interpretation of the alignment results is not a trivial task to assess accuracy. Because of the large amount of multi-mapping, in some of our experiments, we configure aligners to report many alignments per read, to increase the chance of finding the correct alignment in a tradeoff for time. Therefore, to have a practically useful evaluation that can better hide the noise in the alignment and provide a more stable set of results, we calculate the accuracy over the estimated abundances using a quantification tool such as Salmon. In Table 6 the accuracy metrics are calculated over the abundance estimations obtained over the alignments produced running the aligners in different modes. The list of metrics for metagenomic expression evaluations have been chosen similar to previous works such as [99] and [134].

The metrics selected are *Spearman Correlation*, *Mean Absolute Relative Difference (MARD)*, *Mean Absolute Error (MAE)*, and *Mean Squared Log Error (MSLE)*. Each indicating different characteristics of the predicted abundance estimations. For example, lower MARD indicates better distribution of the reads among the references relative to the abundance of each reference, while MAE shows the quality of

Alignment Mode	Tool	Spearman	MARD	MAE	MSLE
Primary	PuffAligner	0.69	0.028	1.39	0.08
	Bowtie2	0.58	0.053	2.91	0.15
	STAR	0.727	0.023	1.493	0.05
	deBGA	0.28	0.616	656.08	6.53
Up to 20	PuffAligner	0.9	0.006	0.40	0.01
	Bowtie2	0.85	0.01	0.22	0.01
	STAR	0.929	0.004	0.303	0.00
	deBGA	0.28	0.573	637.60	5.65
Up to 200	PuffAligner	0.97	0.002	0.36	0.00
	Bowtie2	0.99	0.001	0.19	0.00
	STAR	0.929	0.004	0.299	0.00
	deBGA	0.28	0.571	637.83	5.55
Best strata	PuffAligner	0.97	0.002	0.36	0.00
	STAR	0.929	0.004	0.3	0.00

Table 6: Accuracy of abundance estimation with Salmon using alignments reported by each aligner over different accuracy metrics for the mock sample simulated from a real sample with accession ID SRR10948222. We have ran all the aligners in three main modes; allowing only one best alignment with ties broken randomly (Primary), up to 20 alignments reported per read, and up to 200 alignments. PuffAligner and STAR support a fourth mode that allows reporting all equally best alignments (bestStrata). This option improves the performance while keeping or even slightly improving the accuracy of the results.

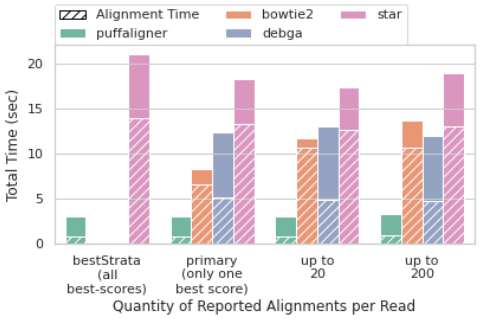
the distribution of the reads in an absolute way regardless of the difference between the abundance of the references. In this case, one misclassified read has the same impact on the MAE metric both for an abundant or low-quantity references. The mathematical definition of each of these metrics is provided in equation 3.2 in the supplementary material.

The three main observations in this experiment are as follows. First of all, regardless of the alignment mode, deBGA reports a vastly underestimated list of counts (considering valid alignments according to the definition). However, PuffAligner, STAR and Bowtie2, show very similar behavior with respect to accuracy. STAR is the best in primary mode as well as allowing 20 alignments closely

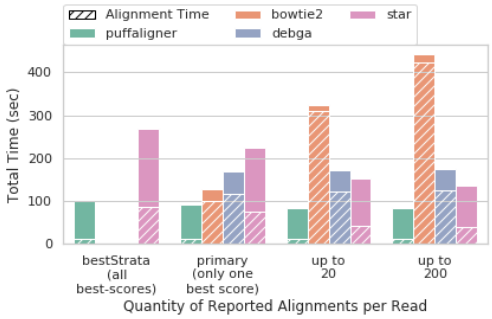
followed by PuffAligner while Bowtie2 is the winner allowing up to 200 alignments again with PuffAligner being the close runner-up. This represents PuffAligner as a reliable alignment tool showing a stable pattern of being comparable to the best aligner in all the cases. Moreover, due to the nature of the metagenomic data – the high amount of ambiguity and multi-mapping – we expect to see improvement in the accuracy metrics as more alignments are reported per read, because, this leads to a higher recall. While STAR's accuracy changes only slightly from 20 alignments to 200 alignments (only improving MAE) the results for PuffAligner and Bowtie2 improve considerably allowing more alignments per read. However, this higher accuracy comes in the cost of alignment time for Bowtie2. As shown in figure 8, section 8b, Bowtie2 alignment time increases allowing more alignments per read while PuffAligner, as a hash-based seed and extend method, exhibits a constant alignment time regardless of number of alignments being reported per read. The difference becomes specifically evident while allowing up to 200 alignments per read, where PuffAligner is 4 times faster than Bowtie2. In addition to all this, in real data, many of the alignments reported do not necessarily have a high quality and only appear in the output as one of the 200 alignments for the read. This leads us to the last but not least observation: the similar accuracy achieved by PuffAligner in *bestStrata* mode compared to when allowing up to 200 alignments. In *bestStrata* mode, PuffAligner limits the reported alignments to only those with the highest score for each read, which in the face of no errors covers all the multi-mapped reads without reporting the sub-optimal alignment as a byproduct. The observations are pretty similar in the other two simulated samples in the supplementary table 11 this

time with PuffAligner being the most accurate aligner beating both Bowtie2 and STAR in different modes for both samples.

Overall, these results along with other similar experiments in supplementary table 11 indicate that PuffAligner is a highly-sensitive and fast aligner. Specifically PuffAligner is an appropriate choice of interest for metagenomic analysis, since it is as accurate as well-known aligners like Bowtie2 and STAR with close memory requirements to Bowtie2, while being much faster.



(a) Time performance for aligning a single strain sample averaged over all three samples.

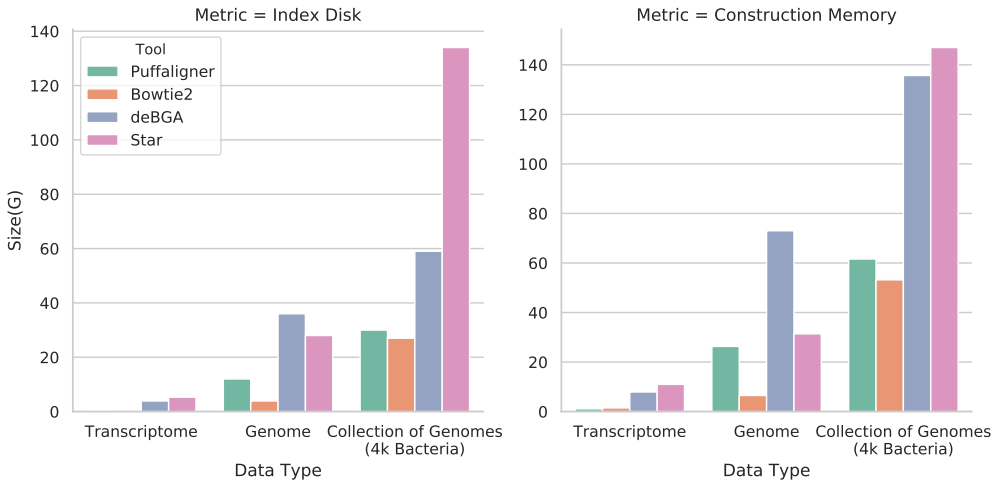


(b) Time performance for aligning a mock experiment simulated from bulk read sample SRR10948222.

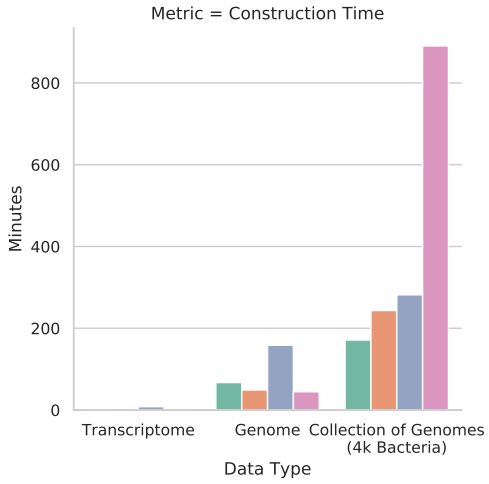
Figure 8: Time performance of different alignments in the two microbiome experiments. In 8a, the results are averaged over the three alignment processes for the samples covid19, sars, and bat2008 each having $\sim 1M$ paired-end reads. In 8b the performance shown is for aligning reads in mock sample simulated from SRR10948222 with $5M$ paired-end reads. As shown in the bulk experiment, the alignment time increases in Bowtie2 asking for more alignments per read while the other tools show a constant alignment time scaling over number of reads. The dashed area shows fraction of the time spent purely on aligning reads where the remaining is possessed by the index loading time. PuffAligner is by far the fastest tool and yet most of its alignment time is dedicated to loading the index. This demonstrates the efficiency of the hash-based alignment methodology in PuffAligner which results its fast alignment time to stand out even more when the index is already loaded in memory.

3.6 Scalability

Figure 9 represents how the construction time and index size of each tool scales over different types of sequences from human transcriptome toward 4000 bacterial



(a) Scalability over index disk space and construction memory



(b) Scalability over construction time

Figure 9: Scalability of different tools over the final index disk space, construction memory, and construction running time for three different datasets, human transcriptome (gencode version 33), human genome (GRCh38 primary assembly), and collection of genomes (4000 random bacterial complete genomes). All tools are run with 16 threads.

genomes. The trend shows the effect of database size as well as redundancy and sequence similarity on the scalability of each of the tools. Tools such as PuffAligner and deBGA, which build a de Bruijn graph based index on the input sequence, specifically compress similar sequences into unitigs and are more prune to the sequence repetition, as a result, these tools are better scalable for databases with high

redundancy such as microbiomes. It is of course necessary to mention that Bowtie2 requires a switch from a 32-base structure to a 64-base one as the total count of the input bases increases which is another reason why the size is growing super-linearly. It is also worth mentioning that since all the aligners require loading the whole index in memory at the time of alignment, since PuffAligner's index is scaling better in the presence of high similarity between reference sequences compared to Bowtie2, the runtime memory requirement for PuffAligner gets closer to Bowtie2.

3.7 Why use an aligner when we have a light-weight and fast pipeline like Kraken2 + Bracken

In this section, we highlight the use case of PuffAligner in the metagenomic world for the upstream step to analyze real data as an accurate, memory-efficient, and fast aligner. Specifically, we approach the problem from the perspective of comparing the correlation of the PuffAligner results, as an aligner, with a k-mer-based abundance estimation approach, namely Bracken. For that purpose, we construct an index over all bacteria, viral, archae, and fungi obtained from NCBI taxonomy database [45] on May 21, 2020 using both Pufferfish and Kraken2. we run both pipelines of Kraken2 +Bracken and PuffAligner +Salmon over 34 randomly selected samples from different categories of metagenomic analyses. Ten of the samples are selected from non-human projects such as the "metasub project" [33] as well as metagenomic samples of submarine or soil analyses [100] and the rest are selected from "human metagenome project (hmp)" [50]. The human samples are chosen from different

tissue categories of plasma, tongue dorsal, gingiva, vaginal, and fecal. We then compare the abundance of the reported references through each of the two pipelines of PuffAligner +Salmon and Kraken2 +Bracken.

We run PuffAligner in two modes; the default mode does not allow any orphan, discordant, or dovetail alignments, Filters any mapping with `minScoreFraction` less than 0.65 of the maximum possible coverage (covering the full read pair) and reports only alignments with highest score (bestStrata). We also run PuffAligner allowing orphans, discordant, and dovetail alignments but keeping the rest of the parameters as default. We use Salmon as a well-known and well-established abundance estimation tool to be able to compare the results with Kraken2 +Bracken reports. It is important to note that the pipeline, although for now being our proposed pipeline for metagenomic analysis on short reads, can be improved by incorporating the specific features of the metagenomic data in abundance estimation step such as the taxonomy tree information and marker genes expression. We also run Kraken2 in two different modes, first the "default" which would allow all reads with even one single k-mer match to be classified and second with setting the confidence option to 0.65 which would prevent reporting reads that have a confidence lower than 0.65. As per authors' definition*, this number is calculated based on the ratio of unique k-mers mapped to a taxa in the taxonomy tree over all the non-ambiguous k-mers of the read (k-mers without "N"s). There is not a one to one correspondence between confidence threshold here and the "minScoreFraction" in PuffAligner. However, both of these options are necessary for providing a more reliable reference abundance report

*https://github.com/DerrickWood/kraken2/wiki/Manual

by removing the reads with small evidence showing the initial sign that the read belongs to the list of indexed references. Filtering orphaned, discordant, or dovetail reads is a feature only available to the alignment-based procedures and not k-mer counting approaches.

Assuming there were no technical errors and no variation across species, if the read was coming from a subset of references in the index, then there would be at least one exact match for it. The edit distance and the scoring criteria for insertion/deletion and mismatch demonstrate the potential technological errors and the variation across individuals in alignment procedure. That along with a threshold for discarding reads with low alignment scores is the established computational approach for deciding which alignment to report. The semi-alignment approaches, break the read into k-mers and instead of full alignment, look for exact k-mer matches, considering the high correlation of this approach and the alignment. This approach is super fast and memory efficient and provides highly correlated abundances with full alignment. However, we believe that in metagenomic analyses this approach is not as highly accurate and correlated as transcriptomic and genomic analyses. The main reason is the variation among genomes goes to its extreme in metagenomics such that two strains could be 99% similar and yet represent two different subtypes of the species.

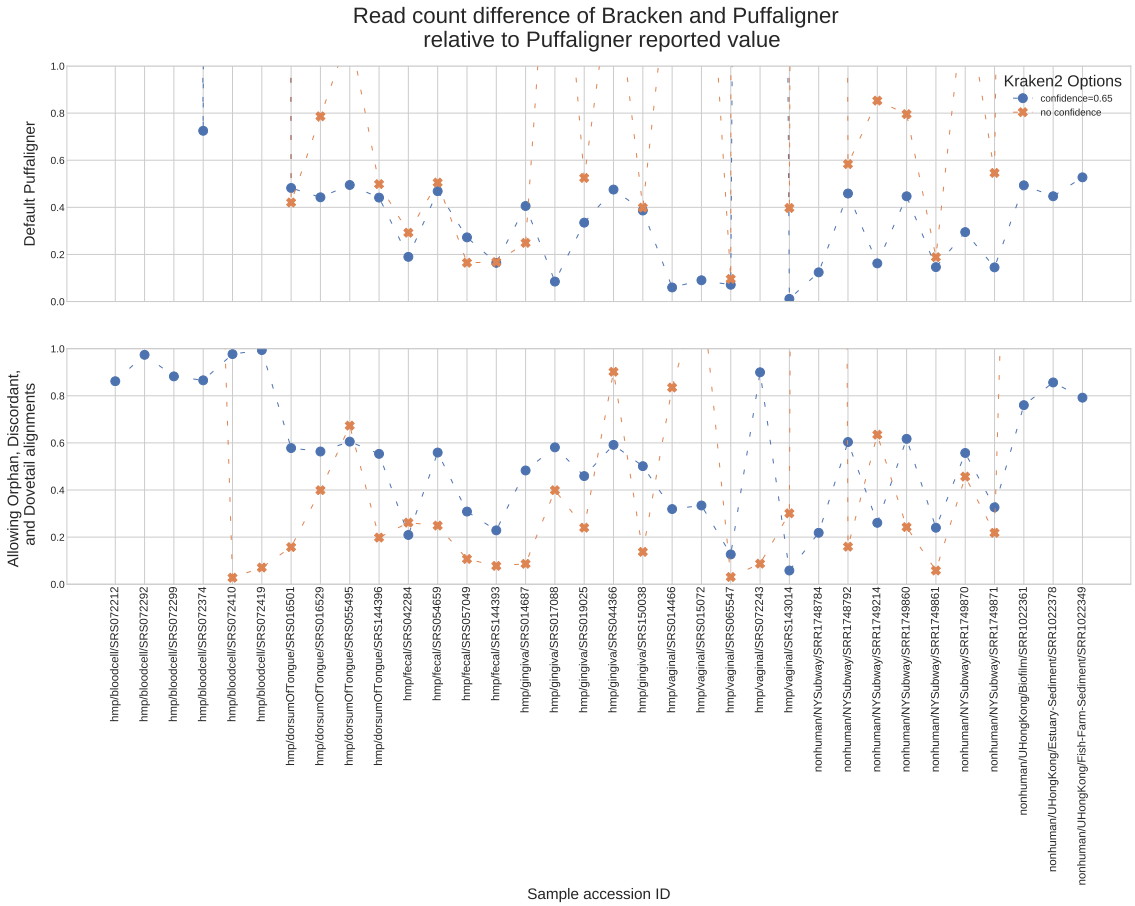


Figure 10: This plot shows, for each of the 34 samples, the difference in count of assigned reads between Puffaligner+Salmon pipeline vs Kraken2+Bracken relative to the reported read count by Puffaligner. The reads are aligned/classified to the Pufferfish/Kraken2 index on the reference sequences of bacteria, viral, archae, and fungi obtained from NCBI. The first and second row compare the results of different Puffaligner runs. In the first row, we run Puffalinger with default parameters and in the second row, Puffaligner also reports orphan, discordant, or dovetail alignments as valid alignments if they pass the alignment score threshold. As the plot shows, there is no well-defined pattern of behavior between Kraken as a kmer counting based classifier and Puffaligner as a full aligner.

Considering the alignment-based as the most representative model for variation and errors in the reads, the plots in figure 10-a show the inefficiency of the semi-alignment approach to be a simplified version of alignment with consistent behavior across individual samples. We run Kraken2 +Bracken pipeline in the two modes described earlier, with applying the confidence of 0.65 and with no conifdence. In figure 10, we compare the effect of this option on the count of accepted

reads with PuffAligner +Salmon reported read count In contrary, we do not see a similar pattern in all the samples, with either the threshold making the reported read counts closer to the alignment pipeline or further from it. It is expected that the same threshold value in the two pipelines of Kraken2 +Bracken and PuffAligner +Salmon, do not provide us with the same read set and therefore a different (higher or lower) threshold might be needed to achieve less read count difference in the alignment and semi-alignment approaches. On the other hand, we should at least see a similar relative effect of applying the threshold over all the samples, i.e. the read count consistently getting closer to the alignment approach or further from it. The options of orphan, discordant, and dovetail reads, although important, are only available to the alignment procedures which consider the relationship of the ends in a paired-end read. That is why we also do the same comparison between the two modes of Kraken2 with PuffAligner discarding the three filtering options so that now the only prohibiting option from reporting a read is related to the reads coverage. The results are shown in the bottom plot of figure 10. Interestingly, the read counts reported in both modes of Kraken2 have got further from the PuffAligner results compared to the plot on top. But the trend of inconsistency in the ratio of the reads reported by Kraken2 over PuffAligner across samples. We provide the absolute read count difference in supplementary figure 10. However, it is not possible to scale the plot so that we can both see the difference for smaller samples and larger ones. We also look at the top 5 highly reported species for each sample and their abundance in supplementary figure 13 for Kraken2 +Bracken in two modes and PuffAligner (default) to see the effect of the pipelines on the list of discovered

species and their abundances. There we similar highly abundant species in Kraken2 with no confidence value as well as PuffAligner, that are not as abundant in Kraken2 with confidence=0.65 for some samples (Streptococcus gordonii for one of the subway samples) whereas for others applying the confidence threshold puts the results closer to PuffAligner (e.g. Lactobocillus Crispatus for two vaginal sample). We do the same analyses at the level of genus in figure 14 and there we observe the same inconsistency. To summarize, through the experiments in this section, we show that the semi-alignment approaches can result in different reported abundant references than alignment approaches per metagenomic sample depending on the quality of the sample and the technological biases particular to that sample. That is why we believe, having a sensitive, and highly efficient alignment pipeline in PuffAligner, it is now a reasonable and required tradeoff of query speed for accuracy to switch from a semi-aligner pipeline to an aligner one for metagenomic analyses.

4 Discussion & Conclusion

In this paper we introduce PuffAligner, an aligner for short read sequences, suitable for the contiguous alignment of short-read sequencing data. We demonstrate its use in aligning single-species DNA-seq reads to the genome, RNA-seq reads to the transcriptome, and multi-species DNA-seq reads to a metagenomic reference. It is built on top of the Pufferfish index, which constructs a colored compacted de Bruijn graph using the input reference sequences. PuffAligner begins read alignment by collecting maximal exact matches, querying k-mers from the read in the Puffer-

fish index. The aligner then chains together the collected MEMs using a dynamic programming approach, choosing the chains with the highest coverage as potential alignment positions for the reads. Finally, PuffAligner is able to efficiently compute the exact alignments, exploiting the information from long matches in the chains.

We compare the accuracy and efficiency of PuffAligner against two widely used alignment methods, Bowtie2 and STAR, that perform unspliced and spliced alignments of reads, respectively. We also compare the results against deBGA, an aligner that also utilizes an index built over the compacted de Bruijn graph.

We analyze the performance of these tools on both simulated and experimental DNA and RNA sequencing datasets. The accuracy of PuffAligner is particularly comparable to Bowtie2, which exhibits both as high sensitivity, and specificity in terms of read alignment, and generally performing better than STAR and deBGA (though, unlike STAR, none of these other tools yet support spliced read alignment). In terms of speed and memory, PuffAligner reaches a tradeoff between the relatively high memory usage of STAR and deBGA and the slower speed of Bowtie2. Hence, while the memory requirement is more than that of Bowtie2, the speed gain is significant, with STAR being the only tested tool that is sometimes faster.

An additional advantage of the Pufferfish index utilized in PuffAligner is that it can be built on a mixed collection of genomes, transcriptomes, or both. We utilize this feature of Pufferfish in a specific pipeline for aligning experimental RNA-seq reads where reads might originate from genomic sequences as well in [152]. The analysis shows how we can improve the specificity of the alignments by discarding potentially intronic reads or reads aligning from processed pseudogenes with similar

sequence to annotated protein coding transcripts. This allows further improving the accuracy of alignments compared to a highly sensitive tool such as Bowtie2. Furthermore, the use of de Bruijn graphs in the design of the Pufferfish index structure and the highly accurate and fast alignment procedure of the PuffAligner makes it particularly useful for indexing and aligning to a highly similar collection of sequences, potentially making it a powerful approach in metagenomic analyses.

We have provided a proof of concept to this fact in our experimental design over a collection of bacteria and plan to specifically use PuffAligner for metagenomic analyses in the future.

5 Supplementary Material

Table 7: The percentage of aligner engine calls skipped in the alignment calculation pipeline.

sample	Cache Hits	Perfect Chains	None Alignable	Total Skipped
DNA-seq experimental	52.89%	19.01%	0.71%	72.67%
RNA-seq simulated	28.69%	50.80%	0.97%	80.46%
Metagenomic simulated	61.10%	31.33%	0.00 %	92.43%

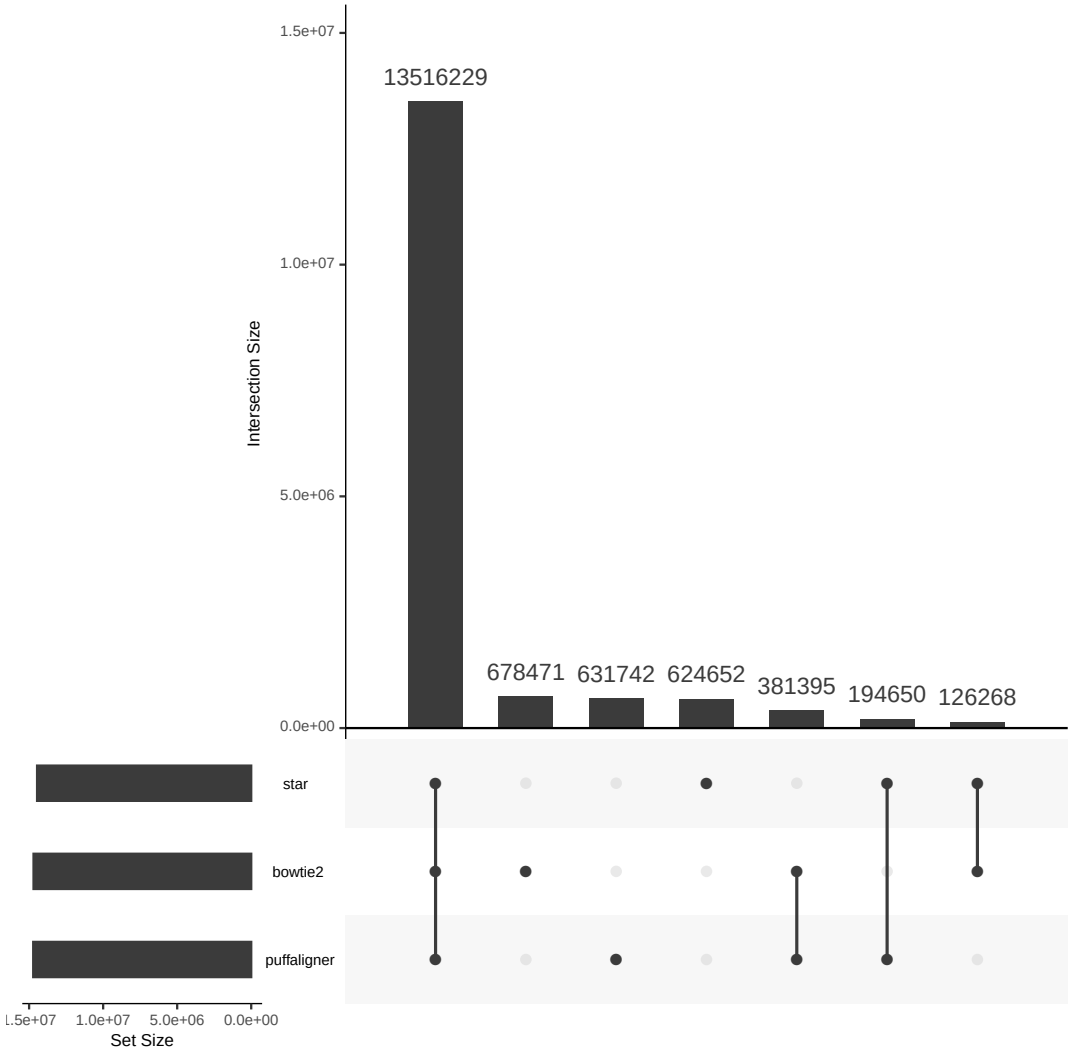


Figure 11: Upset plot showing the agreement of the alignments found by different tools based on the location of the mappings

Table 8: The construction benchmark and final index size for each of the tools over 4000 selected bacteria

Tool	Time (hh:mm)	Memory (GB)	Index Size (GB)
PuffAligner	01:40	61.30	46
deBGA	04:42	129.40	59
Bowtie2	04:03	50.70	27
STAR	14:50	147.70	134

Table 9: Basic information for samples selected for simulating mock bulk metagenomic samples.

Accession	Project description	# of reads	# of reads aligned to 4k selected reference	# of simulated reads	# of references of origin for the simulated reads
SRR10948222	Finding sub-biocrust soil microbial communities in Mojave Desert, California, United States	27,296,270	200k	5,550,650	98
SRR11283975	The impact of different acidification degrees on the bacterial community in the Jiaodong Peninsula, China	35.5k	8,333	1,012,176	92
SRR11496426	The composition, genetics characteristics and structure of the microbial communities of the oil site of Uzon Caldera	42.3k	30,203	1,029,382	179

Table 10: Alignment Distribution for two samples of 500,000 simulated reads from reference sequences NC_004718.3 (known as SARS Coronavirus) and NC_014470.1 (Bat Coronavirus assembled in 2008) respectively.

Sample's Origin	Alignment Mode	Tool	NC_045512.2 (Covid19)	NC_004718.3 (SARS Coronavirus)	NC_014470.1 (Bat Coronavirus)	Other Assembled References
SARS	Primary	PuffAligner	0	500,000	0	0
		Bowtie2	24	499,975	0	0
		STAR	0	499,998	0	0
		deBGA	0	499,995	0	5
	Up to 20	PuffAligner	116	500,000	486	0
		Bowtie2	21,546	499,999	7,205	0
		STAR	0	499,998	0	0
		deBGA	0	499,995	0	5
	Up to 200	PuffAligner	116	500,000	486	0
		Bowtie2	21,546	499,999	7,205	0
		STAR	0	499,998	0	0
		deBGA	0	499,995	0	5
	Best strata	PuffAligner	0	500,000	0	0
		STAR	0	499,998	0	0
Bat2008	Primary	PuffAligner	0	0	500,000	0
		Bowtie2	0	0	499,999	0
		STAR	0	0	499,999	0
		deBGA	0	0	499,991	9
	Up to 20	PuffAligner	32	494	500,000	0
		Bowtie2	2,343	7,127	499,999	0
		STAR	0	0	499,999	0
		deBGA	0	0	499,991	0
	Up to 200	PuffAligner	32	494	500,000	0
		Bowtie2	2,343	7,127	499,999	0
		STAR	0	0	499,999	0
		deBGA	0	0	499,991	9
	Best strata	PuffAligner	0	0	500,000	0
		STAR	0	0	499,999	0

$$
\begin{aligned}
\mathrm{MARD}(y,\hat{y}) &= \frac{1}{n_{\mathrm{refs}}}\sum_{i=0}^{n_{\mathrm{refs}}-1}\frac{|y_i-\hat{y}_i|}{y_i+\hat{y}_i}.\\
\mathrm{MAE}(y,\hat{y}) &= \frac{1}{n_{\mathrm{refs}}}\sum_{i=0}^{n_{\mathrm{refs}}-1}|y_i-\hat{y}_i|\,.\\
\mathrm{MSE}(y,\hat{y}) &= \frac{1}{n_{\mathrm{refs}}}\sum_{i=0}^{n_{\mathrm{refs}}-1}(y_i-\hat{y}_i)^2.
\end{aligned}
\tag{3.2}
$$

Table 11: Alignment accuracy of the tools over different accuracy metrics for the mock sample simulated from real samples with accession IDs SRR11283975 and SRR11496426.

Accession ID	Alignment Mode	Tool	Spearman	MARD	MAE	MSLE
SRR11283975 Jiandong Peninsula	Primary	PuffAligner	0.71	0.024	0.43	0.04
		Bowtie2	0.615	0.04	0.64	0.07
		STAR	0.727	0.02	0.406	0.04
		deBGA	0.274	0.521	106.78	3.79
	Up to 20	PuffAligner	0.942	0.003	0.07	0.00
		Bowtie2	0.909	0.005	0.05	0.00
		STAR	0.946	0.003	0.087	0.00
		deBGA	0.277	0.489	101.39	3.37
	Up to 200	PuffAligner	0.979	0.001	0.07	0
		Bowtie2	0.97	0.002	0.04	0.00
		STAR	0.951	0.003	0.086	0.00
		deBGA	0.278	0.483	100.96	3.29
	Best strata	PuffAligner	0.979	0.001	0.063	0
		STAR	0.951	0.003	0.086	0.00
SRR11496426 Uzon Caldera	Primary	PuffAligner	0.568	0.112	32.55	0.95
		Bowtie2	0.53	0.14	38.06	1.10
		STAR	0.559	0.118	31.823	0.83
		deBGA	0.367	0.566	115.88	3.57
	Up to 20	PuffAligner	0.789	0.03	7.43	0.24
		Bowtie2	0.74	0.042	10.83	0.30
		STAR	0.713	0.049	6.939	0.17
		deBGA	0.368	0.554	109.29	3.32
	Up to 200	PuffAligner	0.865	0.017	5.64	0.11
		Bowtie2	0.879	0.015	7.21	0.13
		STAR	0.724	0.045	6.496	0.13
		deBGA	0.369	0.549	108.99	3.27
	Best strata	PuffAligner	0.85	0.019	5.571	0.09
		STAR	0.723	0.046	6.544	0.13

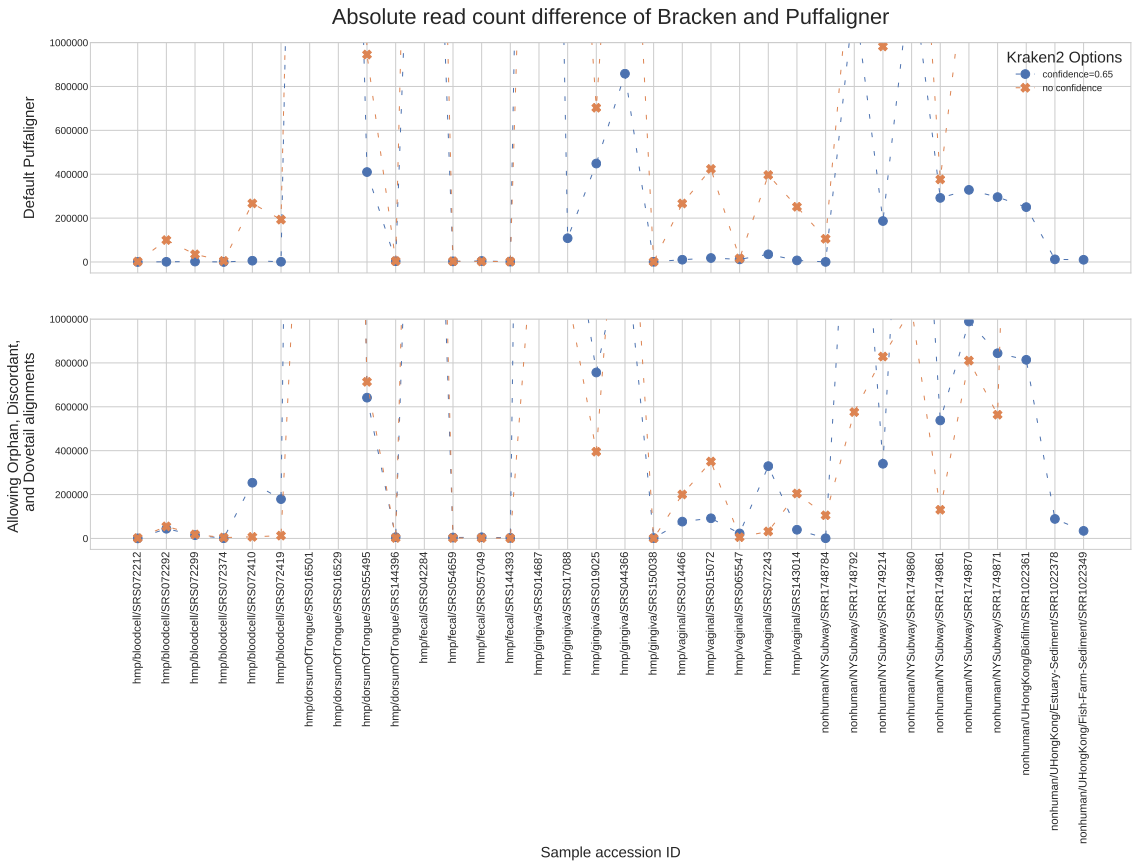
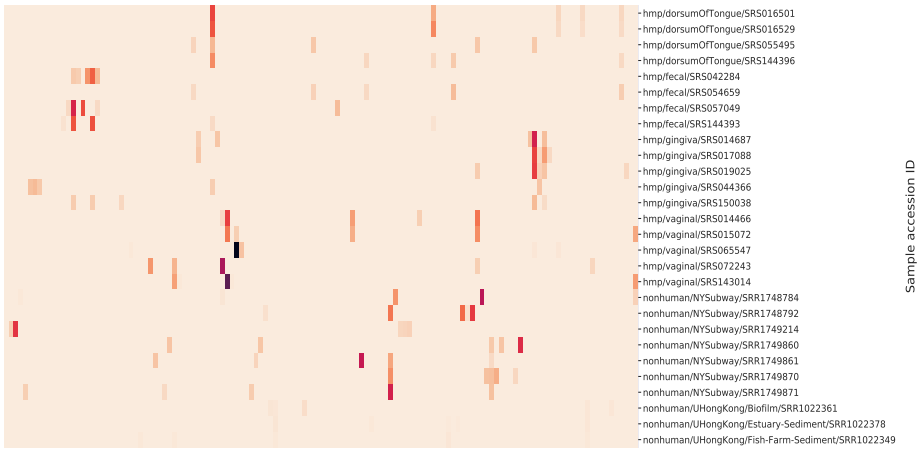
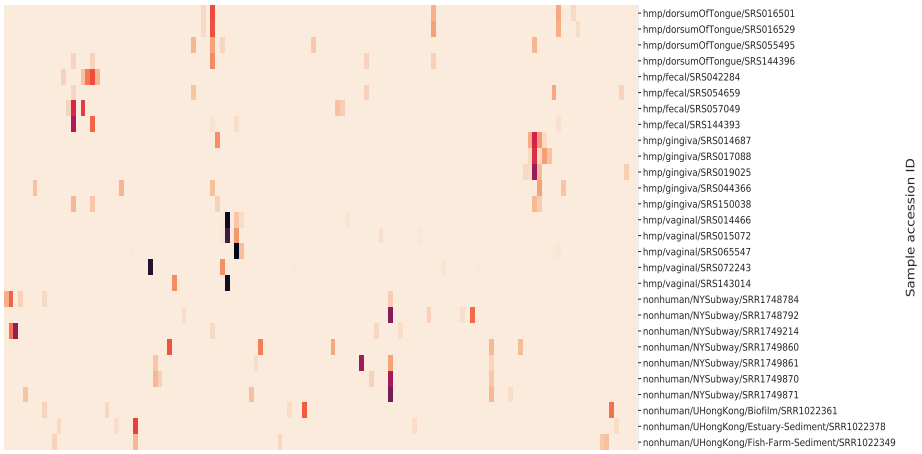


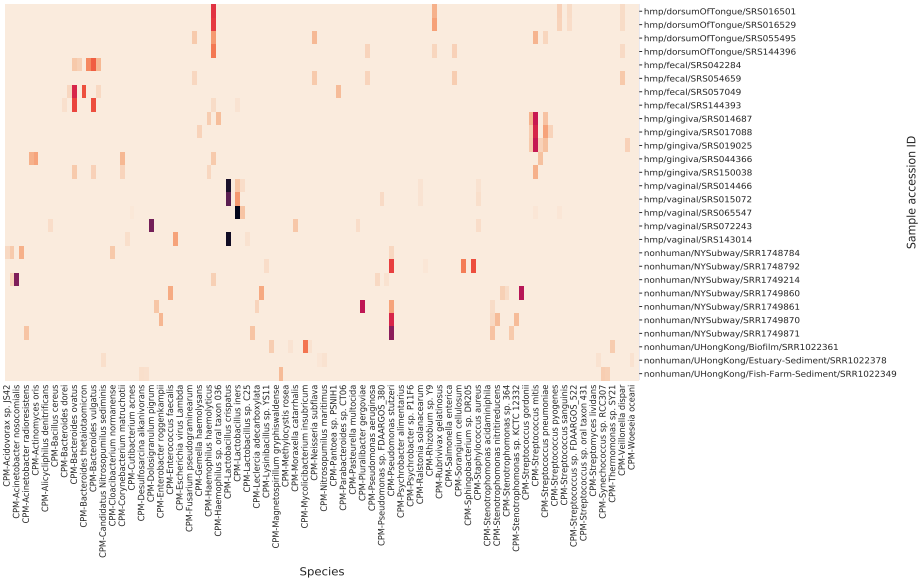
Figure 12: The difference in count of assigned reads between Puffaligner+Salmon pipeline vs Kraken2+Bracken reads on 34 samples. Puffaligner is run in the mode that does not allow any orphan, discordant, or dovetail alignments.



(a) Bracken with no confidence



(b) Bracken with confidence=0.65



(c) Puffaligner default + Salmon

Figure 13: Heatmap showing 5 most popular species over 28 samples through three pipelines of Kraken2(no confidence)+Bracken, Kraken2(confidence=0.65)+Bracken and default Puffaligner+Salmon. Overall, we observe more similarity between Puffaligner and Bracken with confidence of 0.65. However, there are cases where applying the confidence filter to Kraken make the results diverge from Puffaligner pipeline for example "Screptococcus gorbanii" considered abundant in a subway sample in the first and third heatmap, whereas Kraken2(confidence=0.65) does not detect the microorganism as abundant.

(a) Bracken with no confidence

(b) Bracken with confidence=0.65

(c) Puffaligner default + Salmon

Figure 14: Heatmap showing 5 most popular genera over 28 samples through three pipelines of Kraken2(no confidence)+Bracken, Kraken2(confidence=0.65)+Bracken and default Puffaligner+Salmon.

Chapter 4: Rainbowfish: A Succinct Colored de Bruijn Graph Representation [4]*

1 Introduction

This paper proposes a new representation of the colored de Bruijn graph. The colored de Bruijn graph is a variant of the de Bruijn graph where each edge (i.e., k-mer) is associated with some set of colors. Here, each color is used to encode the source of the corresponding k-mers (e.g., different source genomes, transcriptomes, sequenced samples, etc.). From this perspective, it is a flexible and powerful combinatorial structure for representing a collection of sequences while maintaining the identity of each. This structure gained popularity in the work of Iqbal et al. [66], which demonstrated the utility of the colored de Bruijn graph for representing and assembling a collection (population) of genomes, and for detecting both simple and complex genetic variants with high accuracy. Analysis of the colored de Bruijn graph exhibits particular promise for analyzing complex population-level variation, since topological structures (e.g., bubbles) can be associated with variation in the underlying sub-populations. The representation adopted by Iqbal, as implemented in the tool `Cortex`, is optimized for speed, and so requires a considerable amount

*published in WABI2017

of memory to represent both the topology of the de Bruijn graph and the colors associated with each edge.

The memory usage of the colored de Bruijn graph representation adopted in `Cortex` precludes this approach from being adopted when the underlying genomes and color sets become too large. In order to overcome such limitations, Muggli et al. [109] introduced the VARI representation of the colored de Bruijn graph. This approach sacrifices some of the speed of the `Cortex` representation for a considerable reduction in the required space. VARI achieves this space savings in two ways. First, rather than using a hash-table-based representation of the de Bruijn graph topology, it adopts the highly-efficient BOSS representation. The BOSS [18] representation (named based on the initials of the authors) makes use of the FM index [46] to encode the topology of the de Bruijn graph. BOSS uses $4N + o(N)$ bits to represent a de Bruijn graph with N edges (empirically, this often works out to be as few as 4-6 bits per edge).

VARI couples the BOSS representation of the de Bruijn graph topology with a compressed representation of the color information. By its nature, BOSS assigns to every de Bruijn graph edge a distinct rank in the range $[0, N)$. So, VARI represents the color information as a $N \times C$ bit matrix where C is the number of input colors. Conceptually, each of the N rows of this matrix is simply a bit vector that encodes which of the C colors label the corresponding edge. To reduce the space required to store this color information, VARI concatenates these rows into a single vector over $N \times C$ coordinates and stores them in an Elias-Fano [42, 44] encoded bit vector, allowing for a (sometimes substantial) reduction in the size while still enabling

efficient point queries (i.e., is a particular edge labeled with a given color?). Muggli et al. [109] demonstrate that the VARI representation can be built on data sets consisting of large numbers of k-mers, large input color sets, or both. Specifically, the space efficiency of VARI makes it possible to build and query the colored de Bruijn graph on datasets that are orders of magnitude larger than what is possible with `Cortex`. This is an exciting development that opens up this methodology for increasingly large-scale analysis.

Though VARI achieves a substantial improvement in space over `Cortex`, there is still a considerable amount of redundancy present in its representation. Both of these rainbowfishs represent the color set corresponding to each k-mer independently of other k-mers. Hence a considerable amount of redundant information can be present when the color set for each k-mer is represented independently. In fact, some existing colored de Bruijn Graph representations, like the Bloom Filter Trie [63] exploit this redundancy to compress shared color information, and share certain ideas and motivation with the representation proposed in this paper. However, many of the possible subsets of colors do not occur in practice, and there is an inherent (often extreme) skewness in the distribution of the color sets that do appear. It becomes even more important to exploit this skewness for large metagenomic datasets because the space usage of VARI for these datasets can become impractical.

In this paper, we introduce a succinct representation, called Rainbowfish, of the color sets associated to each edge in the de Bruijn graph. We also adopt the BOSS representation of the de Bruijn graph topology, and focus, specifically, on how to concisely represent the color information. Rainbowfish's colored de Bruijn

graph representation is entropy compressed and exploits the high skewness present in the distribution of color sets. By exploiting a more efficient decomposition of the set of present colors (i.e., in terms of equivalence classes), we achieve a considerable reduction over the space required by VARI (up to 20× depending on the dataset), while still retaining efficient (i.e., constant time) queries.

2 Background and definitions

Rainbowfish is a succinct representation of the color information, and uses rank and select operations to lookup the color class corresponding to k-mers in the de Bruijn graph. Here, we briefly recapitulate the definition of a succinct data structure and the rank and select operations.

A ***succinct data structure*** consumes an amount of space that is close to the information-theoretic optimum. More precisely, if Z denotes the information-theoretic optimal space usage for a given data structure, then a succinct data structure uses $Z + o(Z)$ space [69].

rank and ***select*** [69] are operations that are commonly used for navigating within succinct data structures. For a bit vector $B[0, \ldots, n-1]$, $\textsc{rank}(j)$ returns the number of 1s in the prefix $B[0, \ldots, j]$ of B. $\textsc{select}(r)$ returns the position of the rth 1, that is, the smallest index j such that $\textsc{rank}(j) = r$. For example, for the 12-bit vector $B[0, \ldots, 11]$ =100101001010, $\textsc{rank}(5) = 3$, because there are three bits set to one in the 6-bit prefix $B[0, \ldots, 5]$ of B, and $\textsc{select}(4) = 8$, because $B[8]$ is the fourth 1 in the bit vector.

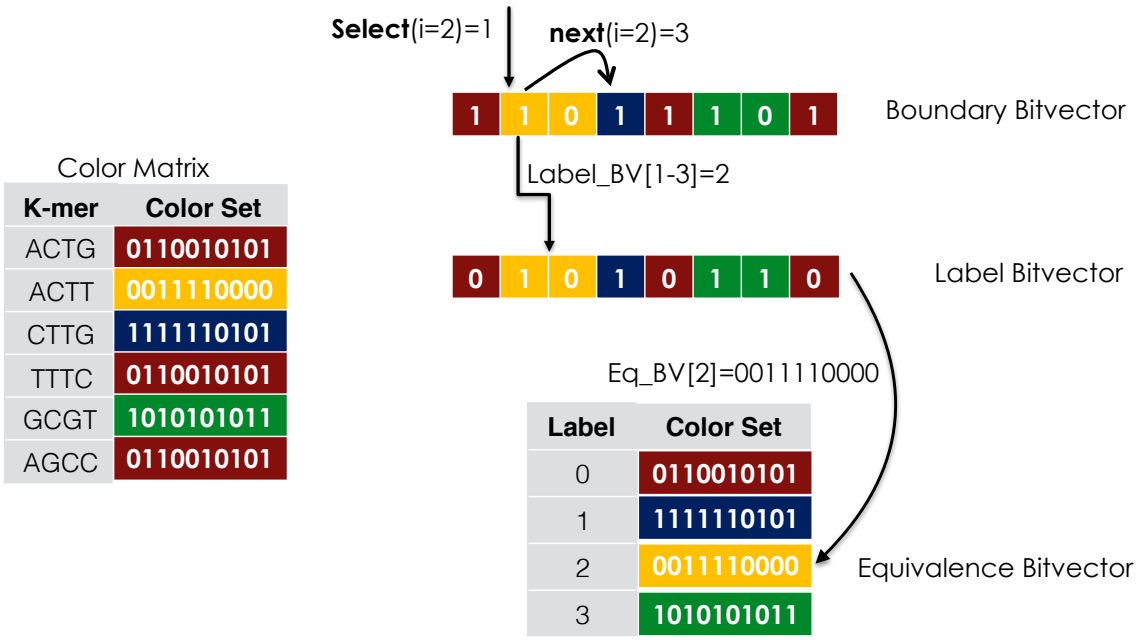


Figure 15: The representation of color information in Rainbowfish. The "Color Matrix" at the top represents 6 distinct 4-mers, each assigned a color set. 3 of these 4-mers (ACTG, TTTC, AGCC) have the same color class, labeled 0, and the other 3 (CTTG, ACTT, and GCGT) each have color classes labeled 1, 10, and 11 respectively. To retrieve the color set for a k-mer, we first perform select on the boundary bit vector (BBV) using rank r of the corresponding edge (k-mer). This returns the label's starting position, i. We then look for the next set bit BBV to find the label's ending position, j. Then, we fetch the label at indices i to j in label bit vector (LBV). Finally, we lookup the label l in the equivalence class table (ECT) and return the color class corresponding to the label. A detailed explanation of the data structure and its construction is given in Section 3.1.

3 Method

In this section we first describe the design of Rainbowfish. We then analyze the space usage and provide a lower bound for the representation of sets of colors given a ranking of de Bruijn graph edges. Finally, we discuss the Rainbowfish implementation.

3.1 Design

Rainbowfish's compact representation of color information is based on two particular observations. First, it is often the case that many of the k-mers in a colored de Bruijn graph share the same set of colors. More formally, we define an equivalence relation $\sim$ over the set of k-mers in the de Bruijn graph. Let $\texttt{Col}(\cdot)$ denote the function that maps each k-mer to its corresponding set of colors. We say that two k-mers are color-equivalent (i.e., $k_1 \sim k_2$) if and only if $\texttt{Col}(k_1) = \texttt{Col}(k_2)$. We will refer to the set of colors shared by the k-mers related by $\sim$ as a ***color class***. If C, the number of input colors, is large, it is often the case that the number of distinct color classes is far less than the number of possible color classes (which is bounded above by $\min(N, 2^C)$).

Second, it is often the case that the frequency distribution of color classes is far from uniform. Hence, it will often be useful to record a frequently occurring color class using a short description (i.e., a small number of bits) while reserving larger descriptions for less frequent color classes.

The design of Rainbowfish is motivated by the above observations. Instead of storing the color set for each k-mer separately, Rainbowfish stores each distinct color class only once and assigns to each distinct class a label (which, practically, is much smaller than the unary encoding of the color class itself). It then stores, for each k-mer, the label of the color class to which it belongs.

The approach we use to assign variable-length labels to color classes is similar in spirit to the construction of a Huffman code, where the message is a string of

color class symbols. However, we do not build a prefix code, and instead opt to store an additional bit vector to allow the efficient selection of an arbitrary label from the list. We generate the labels according to the following procedure. We first sort, in descending order, all the color classes based on their frequency (i.e., the number of k-mers in this color equivalence class). We then assign labels to each color class starting from the class with the largest cardinality, so that the color class represented by the most frequent label will have the shortest label length etc.

The color class representation in Rainbowfish has three components. Rainbowfish stores the mappings between labels and color classes in an ***equivalence class table (ECT)***. As labels are assigned sequentially, this is simply an array of bit vectors encoding the corresponding color sets. Apart from the equivalence class table, Rainbowfish maintains two bit vectors, a ***boundary bit vector (BBV)*** and a ***label bit vector(LBV)***.

All color classes are stored in the equivalence class table (with their corresponding labels implicitly being their position). However, we now need to store a mapping from k-mers to the variable-length labels. Rainbowfish stores variable-length labels corresponding to each k-mer in the label bit vector. The labels are stored in the order in which k-mers are stored in the de Bruijn graph representation. Specifically, the k-mers are stored in the rank order induced by BOSS. However, since these labels are variable-length, we can not directly read the label corresponding to the k-mer of a specific rank, since we do not know where such a label begins or how long it is.

To address this, Rainbowfish maintains another bit vector — the boundary

bit vector (BBV) — to mark the boundary of each variable-length label in LBV. The BBV is the same size as the LBV and has a bit set to 1 at each index where a new label starts in the LBV. Thus, the starting position for the label corresponding to the rth k-mer can be obtained by issuing a SELECT(r) query on BBV, and the length of this label can be obtained by simply scanning BBV until we encounter the next set bit.

Figure 15 shows how the color classes are represented in Rainbowfish. To perform a query for the color class corresponding to a k-mer in the colored de Bruijn graph, we first get the rank r of the k-mer in the de Bruijn graph. We then perform a select operation using r on BBV. The result of the select operation i is the start index of the label of the color class in LBV to which the k-mer belongs. To find the length of the label we determine the index i' of the next bit set in BBV using the TZCNT instruction. TZCNT returns the number of trailing zeros in its argument. If B is a 12-bit vector such that $B[0, 11]$ =110010100000 then TZCNT$(B) = 5$. Using i and i' we retrieve the label from LBV, and using the label we lookup the corresponding color class in ECT. We also note that, as we never have $> 2^{64}$ distinct k-mers in practice, and number of distinct labels is at max equal to the number of distinct k-mers (when each k-mer has a unique label), then we never have $> 2^{64}$ labels. Hence, we can always represent a label using a single machine word. Consequently, we will always reach the next set bit in the LBV after scanning at most a single machine word when starting from current label. This ensures we need only issue a single TZCNT instruction per label decoding call.

3.2 Space analysis

The color class representation in Rainbowfish is entropy compressed, i.e., the space is bounded by the entropy ($H(X_c)$) of the color class distribution. For a dataset in which number of k-mers belonging to each distinct color class are similar, the entropy of the color class distribution will be high. On the other hand, if most of the k-mers in a dataset belong to a small number of distinct color classes, the entropy of the color class distribution will be low.

Lemma 1. *The size of each color class label is bounded by* $\log_2 M$ *bits, where* M *is the total number of distinct color classes. For a dataset with* N *distinct* k*-mers coming from* C *input samples (i.e., colors), we have that* $M \leq \min(N, 2^C)$.

Theorem 1. *Given an ordering of edges (or* k*-mers) in a de Bruijn graph, the space needed by Rainbowfish to represent a set of colors attached to each edge is* $O(MC + NH(X_c))$ *bits, where* M *is the number of distinct color classes,* C *is the number of colors,* N *is the number of distinct* k*-mers, and* $H(X_c) = -\sum_{i=1}^{M} P(x_i) \log P(x_i)$ *is the entropy (i.e., order-*0 *or Shannon's entropy) over random variable* X_c*, which distributed according to the frequency distribution of the color classes.*

Proof. The space needed by Rainbowfish can be analyzed as follows. There are three bit vectors in Rainbowfish, the equivalence class table, label bit vector, and boundary bit vector. To store an equivalence class table containing M distinct color classes each having C colors we need MC bits. To store a label bit vector (as stated in Lemma 1), for N k-mers, where each label corresponds to one of the M

distinct color classes, takes $N \log_2 M$ bits. However, as explained in Section 3.1, in Rainbowfish we assign (optimal) variable-length labels based on the frequency of color classes. Therefore, the space needed to store the label bit vector is dependent on the 0th-order entropy of the color class variable, $H(X_c)$, and the size of the label bit vector is upper bounded by $N \log_2 M$. The boundary bit vector has the same number of bits as the label bit vector. □

3.3 Lower bound for color representation

We now provide a lower bound to store a color class representation for a set of edges in a colored de Bruijn graph. In the color class representation, the equivalence class table takes MC bits to store M bit vectors each having C bits, which is optimal. The other two bit vectors, the boundary and label bit vector, map k-mers given an ordering in the de Bruijn graph to their corresponding color classes. The theorem below gives the lower bound to store such a mapping.

Theorem 2. *The lower bound to represent a mapping from an ordered list of k-mers in a de Bruijn graph to a set of color classes is $\log_2 (M^{N-M} \cdot M!)$ bits, where M is the number of distinct color classes, N is the number of edges, and for a dataset with N distinct k-mers coming from C input samples (i.e., colors), we have that $M \leq \min(N, 2^C)$.*

Proof. We can analyze the lower bound using a counting argument. We count the number of ways to map a set of M distinct color classes to a set of N edges. The space required to store the color class representation should be less than or equal to

the space required to store these mappings.

Edges can be mapped to color classes using a surjective (onto) function. Thus, we wish to count the total number of surjections from M color classes to N edges. Rather than counting this number exactly, we instead provide a lower bound. First, we must ensure that each of the M color classes maps to at least one edge — so, we select a set of M edges and label each with a distinct color class. There are $M!$ ways to assign M color classes to a set of M edges. We will then allow the remaining $N - M$ edges to be colored in any possible manner. We can assign M colors to $N - M$ edges (the remaining number) in M^{N-M} ways. Therefore, the total number of different mappings is bounded below by $M^{N-M} \cdot M!$. To be able to represent each such mapping, and distinguish it from the others, we need at least $\log_2 (M^{N-M} \cdot M!)$ bits. □

The lower bound can be expanded using Sterling's approximation as

$$(N - M) \log_2 M + M \log_2 M - 0.44M + O(\log_2 M),$$

which, ignoring the additive term $O(\log_2 M)$, is greater or equal to $N \log_2 M - 0.44M$. Given the range of M (i.e., $1 \leq M \leq N$), $N \log_2 M$ always dominates the lower bound.

Now, we show that the space needed by Rainbowfish to store the variable-length labels assigned to color classes is equal to the lower bound. As explained in Lemma 1, the upper bound to store any label is $\log_2 M$ bits, and for N edges, it is given by $N \log_2 M$ bits. Rainbowfish also stores a boundary bit vector which has

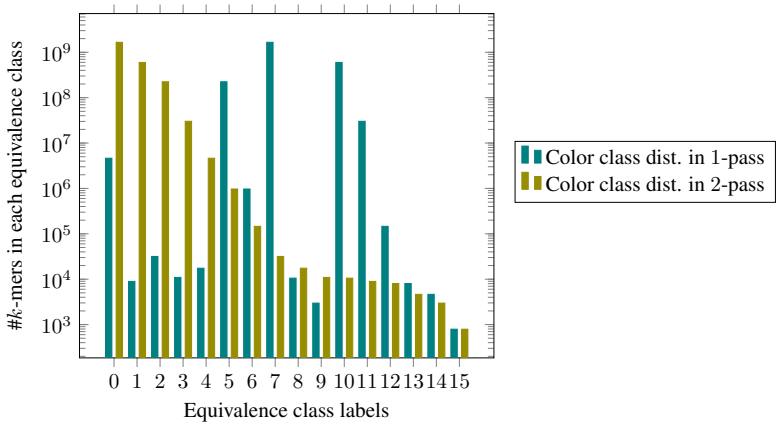


Figure 16: Distribution of k-mer frequencies across equivalence class labels in Rainbowfish after 1-pass and 2-pass algorithm on plant dataset Table 12. The 2-pass algorithm assigns the smallest label to color class with maximum number of k-mers. The distribution in 2-pass algorithm is monotonically decreasing.

the same number of bits as the label bit vector. Therefore, the space required to store the label mappings is strictly $\leq 2N \log_2 M$. Note that the extra overhead to store the metadata to perform a select operation in constant time on the boundary bit vector is bounded by $o(N)$, where N is the numbers of bits in the bit vector [53].

However, Rainbowfish's representation of color classes is entropy compressed (see Section 3.1) and the space required depends on the entropy of the color class distribution. For a highly skewed distribution, the entropy is low and the space required to store labels is much smaller than $N \log_2 M$ bits. On the other hand, when the distribution is near-uniform, i.e., the entropy is high, Rainbowfish makes all labels to be $\log_2 M$ bits and dispenses with BBV. Therefore, the space required by Rainbowfish is always smaller than or equal to the lower bound.

3.4 Implementation

Considerations due to the underlying de Bruijn graph representation We recall here that we make use of the BOSS representation of the underlying de Bruijn

graph topology. To build the BOSS representation, k-mer counting is first performed using KMC2 [38], canonicalizing k-mers during counting. Though the BOSS representation inserts both forward and reverse complement k-mers into the graph, it associates only a single color vector with this pair. Moreover, BOSS creates "dummy" edges (real k-mers prepended or appended with \$) to allow encoding k-mers that appear near terminal nodes in the de Bruijn graph. In the colored de Bruijn graph these dummy edges are assigned the empty color set. All of this information is encoded by both VARI and Rainbowfish. However, as we discuss in more detail in Section 5, the Rainbowfish representation can work with any de Bruijn graph representation that can assign distinct ranks to each k-mer in the de Bruijn graph. Thus, we would expect this encoding scheme to work well with, e.g., a de Bruijn graph representation based on minimum perfect hashing of the k-mers [41].

Storing bit vectors. In Rainbowfish, we use bit vector implementations from the SDSL library [51] to store the three bit vectors from Figure 15. We use the *rrr_vector* implementation from SDSL to store the equivalence class table and boundary bit vector, and the *bit_vector* implementation from SDSL to store the label bit vector.

The *rrr_vector* of SDSL is an implementation of RRR encoding [132]. RRR encoding is an entropy compressed encoding and also supports constant time rank and select operations on the compressed bit vector. The space reduction depends on the entropy of the bit vector. For high entropy bit vectors, the compression is not noticeable and in fact "negative" in some cases because of the extra metadata

overhead to support rank and select operations.

The equivalence class table and boundary bit vector often have fairly low entropy, and can be compressed efficiently using RRR encoding. However, the label bit vector often has high entropy, and compressing it using RRR encoding is not effective. In our representation, the average order-0 entropy of the label bit vector for four different datasets is 0.94. This is a quite high, and hence we did not see any reduction in the space using RRR encoding. However, for the other two bit vectors, the order-0 entropy is lower (e.g., for boundary bit vector the average entropy over same four datasets is 0.56) and, in practice, we achieve a considerable space reduction using RRR encoding.

Construction. We use a 2-pass algorithm to construct the three bit vectors. In the first pass, we read the color matrix, compute the distinct color classes, and count the frequency of each class. Once we have the frequency information, we sort color classes in descending order based on their frequency. We then assign labels to color classes starting from zero. In the second pass, we read the uncompressed color matrix again, and add the label of each k-mer to the label bit vector. While building the label bit vector, we also build the boundary bit vector by storing a 1 at every index where a new label starts in the label bit vector. The labels are stored in the same order as the k-mers in the BOSS representation.

To reduce the space required for the labeling even further, we implemented our label encoding in the following way. Every time that the label size increases from x bits to $x+1$ bits, we restart the counter of that label in label bit vector to 0.

For example, we store 0 and 1 for labels 0 and 1 respectively, then we store 00, 01, 10 and 11 for labels 2, 3, 4 and 5 respectively. For label value 6 we again restart the counter to 0 and store 000 to represent 6 in the label bit vector, etc. Later, when we want to retrieve the actual value of a label, we first recover the stored label l' from the label bit vector and then calculate the actual label l using the equation $l = l' + 2^d - 2$ where d is length of label l in bits.

As explained in Section 3.2, the 2-pass algorithm minimizes the space used to represent color class labels by sorting the classes based on their frequencies and assigning labels to color classes to minimize the length of the resulting code path, similar to Huffman coding. However, one could also imagine assigning labels to color classes as we see them in the order k-mers appear in the BOSS representation. This way, we can construct all three tables in a single pass (i.e., a 1-pass algorithm).

However, as shown in Figure 16, this 1-pass algorithm can end up assigning long labels to frequent k-mers, and hence produce poor (i.e., large) encodings. However, the 2-pass algorithm always assigns labels according to the corresponding frequency distribution of the color classes. Sometimes, the 1-pass algorithm does well, but we chose to adopt the 2-pass algorithm in Rainbowfish.

4 Evaluation

In this section we evaluate Rainbowfish, and compare it to VARI [108], a state-of-the-art colored de Bruijn graph representation. We evaluate both rainbowfishs in terms of space and running time. We address the following questions about the

Datasets	# of edges	# of colors (samples)	# of distinct color classes
E. coli 10	28,273,951	10	479
E. coli 1000	157,737,064	1000	2,669,157
E. coli 5598	435,705,390	5598	7,000,715
E. coli 1000 (k=63)	258,893,268	1000	2,530,253
Plant	2,520,140,426	4	16
Beef safety	97,096,576,010*	87	623,022,532
Human transcriptome	159,441,804*	95,146	340,762

Table 12: The number of edges (include k-mers and dummy edges in the BOSS representation), samples and color classes for different datasets used in the experiments. $k = 32$ unless otherwise specified. *# of edges excluding dummies.

Datasets	Construction Time (secs)		Bubble Calling Time (secs)	
	VARI	Rainbowfish	VARI	Rainbowfish
E. coli 10	44	31	344	366
E. coli 1000	340	270	2,610	2,356
E. coli 5598	3,141	4,021	8,796	8,201
Plant	108	339	47,040	48,537
Beef safety	15,378	30,478	NA	NA
Human transcriptome	13,961	30,804	NA	NA

Table 13: Construction and bubble calling time for Rainbowfish and VARI for different datasets.

performance of Rainbowfish: How does Rainbowfish compare to VARI in terms of the space required to represent color information?; How does Rainbowfish compare to VARI in terms of the construction time?; How does Rainbowfish compare to VARI in terms of typical queries (e.g., in bubble calling)? We are particularly concerned with ensuring that Rainbowfish produces small encodings of the color information and remains practically efficient to query.

4.1 Experimental setup

To answer the above questions, we perform two different benchmarks. First, we evaluate the time taken to construct the color class representation. The construction time is the time taken to construct the color class representation from a list of color

classes stored in the order of the edges in the de Bruijn graph (this is the same input used by VARI). During construction, we adopt a two-pass algorithm. In the first pass, we use a sparse hash-table to determine the distinct color classes and the cardinality of each such class.

We note that the space taken in this first pass is within a small constant factor of the final space required by the final ECT table itself, since we need only store each color class once in the hash table (as a key), and store the associated count (a machine word) as the value. Thus, the memory required by this first pass is almost always a small fraction of the total memory usage of the construction algorithm.

Given this information, we know exactly the number of bits that will be required to store the label and boundary vectors. In the second pass, we fill in both the label and boundary vectors and then save all three structures to file. As with most succinct representations, the space required for our data structure in memory and on disk is almost the same (as the two-pass algorithm allows us to allocate only the space we need for our final representation). The construction time recorded here does not include (for either Rainbowfish or VARI) the time taken to build the de Bruijn graph and color list corresponding to edges in the de Bruijn graph (since this is the same for both methods).

We also report the space needed by both Rainbowfish and VARI to store the color class representation on disk. We do not include the space needed to represent the actual de Bruijn graph in our space comparisons because both Rainbowfish and VARI use BOSS to store the actual de Bruijn graph, and the BOSS representation itself tends to take less space than the color information.

Second, we evaluate the time taken to perform the bubble calling benchmark as described in [109], using both the VARI and Rainbowfish representations. Finding bubbles in a colored de Bruijn graph enables one to detect regions in the de Bruijn graph where different samples (i.e., colors) diverge from each other. As originally suggested by Iqbal et al. [66], such algorithms can form the basis for analyzing certain types of genetic variants in populations of genomes. We note that we adopt the exact bubble calling algorithm implemented in VARI, and the only variable being altered in our bubble-calling benchmark is the data structure being used to determine the set of colors present for each k-mer. Since VARI and Rainbowfish are both built upon the BOSS representation, which is based on the edge-centric view of de Bruijn graph, they consider k-mers as edges in the de Bruijn graph, meaning that each edge is associated with a k-mer, and its corresponding rank and color set. Briefly, the bubble calling algorithm takes as input a pair c_1, c_2 of colors and traverses edges in the de Bruijn graph to find bubbles in which the edges in one sub-path are colored with c_1 and the edges in the other sub-path are colored with c_2 (see [109] for further details).

For all experiments in this paper, unless otherwise noted, we consider the k-mer size to be 32 to match the parameters adopted by Muggli et al. [109]. We carry out these benchmarks on a number of datasets as described in Sect. 4.2. The time reported for construction and bubble calling are averaged over two runs, and the time is measured as the wall-clock time using the `/usr/bin/time` executable. All experiments were performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4

ATA HDD running ubuntu 16.10, and were carried out using a single thread. We note that, while the construction of the color set representation in Rainbowfish (and VARI) are serial operations, queries are trivially parallelizable, as each label can be queried and decoded independently.

4.2 Data

We run our benchmarks on the datasets mentioned in Table 12. The first three datasets, *E. coli*, Plant, and Beef safety are slight variants of those used for evaluation in VARI [109]. Each of these data sets exhibits different characteristics in terms of the number of k-mers, the number of input samples (i.e., colors) and the homogeneity of the underlying samples (i.e., how different are the de Bruijn graph for each of the individual samples). The first dataset consists of the assemblies of 5,598 different strains of *E. coli* obtained from GenBank [116]. Here, each "color" represents a specific *E. coli* assembly. Since these assemblies are from different strains of the same species, they exhibit a small degree of heterogeneity. In other words, a large fraction of the union de Bruijn graph is expected to occur in all samples.

To evaluate the scalability of Rainbowfish when primarily changing the underlying number of input colors, we have evaluated three variants of the *E. coli* dataset. These consist of a dataset containing only 10 different strains, another containing 1,000 different strains and the final containing all 5,598 strains. We also performed experiments with k-mer size to be 63 for *E. coli* 1000 dataset to evaluate the space usage for higher k-mer sizes.

The second dataset (i.e., Plant) consists of the genome assemblies of four different plant species. Hence, this dataset contains only four colors, but has more than $\approx$ 2 billion distinct k-mers. The plant species considered are, *A. thaliana* [*] [155], Corn[†] [142], Rice[‡] [156], and Tomato[§] [25]. These genomes exhibit considerable diversity and heterogeneity. Given the diverse regions in the colored de Bruijn graph, this dataset is a good candidate for the bubble calling benchmark. Further, Muggli et al. [109] found that this was the only of the three original datasets on which they were able to construct the original `Cortex` representation of the colored de Bruijn graph. They validated `Cortex` produces the same bubble calls as VARI [109] (which, of course, produces the same bubble calls as Rainbowfish). For more detailed analysis of `Cortex`'s construction and processing time and space on this dataset, please refer to [109].

The third dataset, Beef safety, is considerably different from the prior data. Instead of the input samples consisting of assembled genomes, they consist of 87 metagenomic samples sequenced from cattle in the commercial process of beef production [115]. Hence, this dataset yields a considerably larger and more complex de Bruijn graph since it is built upon many un-assembled (and non-error-corrected) reads. Thus, the de Bruijn graph will encode portions of the relevant metagenomes as well as the effects of sequencing errors. This dataset also has many more k-mers

[*] ftp://ftp.ensemblgenomes.org/pub/plants/release-34/fasta/arabidopsis_thaliana/dna/Arabidopsis_thaliana.TAIR10.dna.toplevel.fa.gz

[†] ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/005/GCF_000005005.1_B73_RefGen_v3/GCF_000005005.1_B73_RefGen_v3_genomic.fna.gz

[‡] http://rice.plantbiology.msu.edu/pub/data/Eukaryotic_Projects/o_sativa/annotation_dbs/pseudomolecules/version_7.0/all.dir/all.con

[§] ftp://ftp.solgenomics.net/tomato_genome/assembly/build_2.50/SL2.50ch00.fa.tar.gz

than the others, $\approx$ 97 billion. It exhibits a large degree of heterogeneity and an intermediate number of input colors (87).

In addition to the three datasets used in the VARI paper, we also consider building the colored de Bruijn graph on the human transcriptome* (Gencode v26 protein coding transcripts) [60]. Here, we consider each transcript as an individual sample (i.e., a distinct input color). This data consists of $\approx 95,000$ colors, but only $\approx$ 159 million k-mers. Hence, this dataset will give an idea about how the representations will perform when the number of colors becomes very large (though the number of distinct color classes remains orders of magnitude smaller than the number of k-mers). Further, we note that this dataset highlights some of the similarities between the color class encoding adopted by Rainbowfish and the k-mer-based equivalence class decomposition adopted by certain transcript quantification methods (e.g. [126]).

4.3 Performance

Table 13 shows the time taken by Rainbowfish and VARI to construct the color class representation for different datasets. Rainbowfish uses a 2-pass algorithm to construct the color class representation, and hence the construction time is dominated by the steps to read the color list file twice. For small datasets like *E. coli* 10 and *E. coli* 1,000, the input file size is small and does not affect the overall construction time compared to VARI. However, for large datasets like Plant and Beef safety, the

*ftp://ftp.sanger.ac.uk/pub/gencode/Gencode_human/release_26/gencode.v26.pc_transcripts.fa.gz

time to read the color file twice dominates the construction time and Rainbowfish is 1.9×—3× slower. We note that this time can be considerably reduced by avoiding the uncompressed color matrix representation currently used upstream of Rainbowfish and VARI, and integrating determination and encoding of the color classes into the de Bruijn graph construction directly. However, this is outside the scope of the current paper.

Space Table 14 shows the space usage of Rainbowfish and VARI for the different datasets we consider. Among these data, there are a range of characteristics in terms of the number of k-mers, the number of colors, and the complexity and heterogeneity of the de Bruijn graph. We find that, for all datasets, Rainbowfish requires less space to store the color information than VARI. The magnitude of the improvement depends on the number of distinct equivalence classes and their distribution, but is as large as $\sim 20\times$. We see the same trend with higher values of k-mer sizes.

In particular, Rainbowfish's space usage is particularly impressive for datasets with a large number of input colors but a relatively small number of distinct k-mers. In this case, we usually find that the number of distinct color classes is very small compared to the universe of possibilities, and so each label can be encoded in much fewer than C bits. However, the space VARI consumes depends greatly on the sparsity of the color matrix. The color matrix itself grows rapidly as the number of k-mers and colors increases, but VARI's compression mechanism (Elias-Fano encoding) is very effective if the color matrix is sparse (e.g., each k-mer is labeled with only a small subset of colors). This is exactly the case for the Human

Datasets	uncompressed color matrix	VARI	Rainbowfish
E. coli 10	34	58	20
E. coli 1000	18,804	8,848	475
E. coli 5598	290,761	58,718	2,938
E. coli 1000 (k=63)	185,669	8,872	637
Plant	1,202	1,603	497
Beef safety	1,007,009	210,998	144,564
Human transcriptome	1,808,435	841	817

Table 14: The space required by Rainbowfish and VARI to store the color class representation for different datasets. The first column shows space required for the uncompressed color matrix ($N \times C$ bits). All space is reported in MB. $k = 32$ unless otherwise specified.

transcriptome, where the color matrix has an entropy of ~ 0.0004 (compared to *E. coli* 5,598 and *E. coli* 1,000 with entropies of ~ 0.16 and ~ 0.34 respectively). Thus, in the *E. coli* dataset, VARI can save space up to a factor of ~ 5 compared with the uncompressed representation, while in the Human transcriptome it can save a factor of $\sim 2,150$ because of the low entropy of the color matrix. Rainbowfish does well in all experiments, even when the number of input colors is small (e.g., in the Plant dataset). Rainbowfish achieves the most impressive compression when the color class distribution has low entropy and the number of color classes is small relative to the upper bound. In such cases, the entropy compressed representation of Rainbowfish is able to represent a large fraction of all labels using a very small number of bits.

Bubble calling Table 13 shows the time taken by Rainbowfish and VARI to perform the bubble calling benchmark on different datasets. We run the bubble calling benchmark on the *E. coli* and Plant datasets (as in the VARI paper). We note that the current bubble calling algorithm is too slow to run on the Beef safety data set (the time in [109] was estimated at $> 3,000$ hours). It is possible, however, that

optimizations to the underlying algorithm might lift this restriction. We also did not perform bubble calling on the human transcriptome dataset as here, we were unable, given the resources on our server, to even run the de Bruijn graph construction to completion. Specifically, due to the large amount of external memory that VARI uses to build the uncompressed color matrix and the de Bruijn graph on these larger (either in terms of the number of k-mers, the number of colors, or both) datasets (on order of Terabytes), we exhausted the available disk space. For these datasets, to approximate the relevant sizes and construction times, we produced a uncompressed color matrix that lists the colors for each k-mer and its reverse complement, and we use this to build both the VARI and Rainbowfish color representations. While very similar to the full color matrix that VARI would produce, this file is slightly different in that it does not include entries for dummy edges (a detail of the BOSS representation), and the order of the color matrix rows can be different from what will appear in the BOSS representation. However, we still believe these numbers, provided in Table 12, give a reasonable approximation of how the respective methods would perform were we able to construct the de Bruijn graph completely.

For bubble calling, both representations require a very similar amount of time. This is likely due, in part, to the fact that navigating the BOSS representation of the de Bruijn graph may be the performance bottleneck in the bubble calling algorithm. Thus, both VARI and Rainbowfish provide sufficiently fast access to the color sets for each edge that they do not represent bottlenecks in this regard.

5 Discussion & Conclusion

In this paper, we propose an entropy-compressed, succinct data structure to store the color information of a colored de Bruijn graph. To represent the topology of the de Bruijn graph itself, we adopt the BOSS [18] representation. However, we note that, for our representation of the color sets, we only require that the underlying de Bruijn graph representation is able to associate a unique rank between 0 and $N-1$ with each edge. Hence, it is possible to use the Rainbowfish representation with other representations of the de Bruijn graph topology (e.g., those based on minimal perfect hashing).

We demonstrate that the inherent skewness in the distribution of color classes can be exploited to reduce the size of the color information. This allows Rainbowfish to represent the colored de Bruijn graph, even for large datasets with many colors, in a reasonably small space. In fact, for representing the color information itself, we show that Rainbowfish is succinct, and hence requires only $Z + o(z)$ bits where Z is the number of bits required by an information-theoretically optimal representation. Moreover, it may be possible for the color information stored in the equivalence class table to be further compressed to reduce the space. For example, one could imagine an encoding of color sets that takes advantage of their shared subsets, e.g., storing the shared prefixes of membership vectors only once.

While we have described here a rainbowfish for efficiently representing the color information in a colored de Bruijn graph, our encoding scheme can be generalized to store any type of attribute attached to the edges. For example, one could use

the same (or a related) scheme to encode information like the k-mer count or set of positions associated with a given edge. Moreover, it will be interesting to explore how multiple attributes could be efficiently stored simultaneously, and how potential correlations between these attributes might be exploited. For example, there may be natural extensions of similar coding schemes to the *compacted* de Bruijn graph, where one might also be able to take advantage of the coherence in annotation (i.e., color or count information) shared among the constiuent k-mers of a contig, allowing one to store only the information where these annotations change during traversal.

Finally, in our current implementation, the input to the rainbowfish is a color matrix file generated by VARI. This implementation requires first building the uncompressed color matrix, and then permuting the rows of this matrix along with the edges of the de Bruijn graph during the BOSS construction procedure. This process can require a large amount of space, as the uncompressed color matrix can become extremely large (on the order of Terabytes for some of the datasets we considered here). Consequently, in most cases, the construction algorithm must resort to making extensive use of external memory (i.e., disk), which increases building time and consumes a large amount of disk space. However, we note that the Rainbowfish representation can be built without direct access to the uncompressed color matrix.

Specifically, the current VARI algorithm uses a mergesort-like approach to construct the uncompressed color matrix, where the k-mers in each sample are sorted lexicographically (independently), and the rows of the color matrix are constructed one by one by asking for each k-mer, in lexicographic order, which samples contain it. The working memory of this approach is very small compared to the size of

the full color matrix itself. One could imagine using the same merge-based scheme to construct the Rainbowfish representation directly. In the first pass, the distinct color classes and a counter for each would be stored, resulting in a small, sparse hash table rather than a large, uncompressed color matrix. In the second pass, one would simply associate the relevant labels, rather than uncompressed color vectors, with each edge. This would vastly reduce the time and space required to construct the colored de Bruijn graph.

Thus, in the future, we are interested in both incorporating the Rainbowfish representation more tightly inside the existing VARI codebase, as well as pairing the Rainbowfish representation with other compatible representations of the de Bruijn graph topology.

Chapter 5: Mantis: An Efficient, Scalable and Exact Representation of High-Dimensional Color Information Enabled via de Bruijn Graph Search [9, 124]*

1 Introduction

The colored de Bruijn graph (colored de Bruijn graph) [68], an extension of the classical de Bruijn graph [30, 129, 130], is a key component of a growing number of genomics tools. Augmenting the traditional de Bruijn graph 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 de Bruijn graph enables it to be used in a wide range of applications, such as large-scale sequence search [20, 123, 149, 150, 153] (though some [149, 150, 153] do not explicitly couch their representations in the language of the colored de Bruijn graph), population-level variation detection [5, 63, 109], traversal and search in a pan-genome [63], and sequence alignment [95]. The popularity and applicability of the colored de Bruijn graph has spurred research into developing space-efficient and high-performance data-structure implementations.

*A collaboration with members of the Mantis team with two published papers in RECOMB2018 and RECOMB2019

An efficient and fast representation of colored de Bruijn graph requires optimizing both the de Bruijn graph and the color information. While there exist efficient and scalable methods for representing the topology of the de Bruijn graph [19, 28, 30, 36, 122, 139] with fast query time, a scalable and exact representation of the color information has remained a challenge. Recently, Mustafa et al. [113] has 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 colored de Bruijn graph with even a moderate number of input samples (see Figure 19b). 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. introduced colored de Bruijn graphs [68] and proposed a hash-based representation of the de Bruijn graph in which each k-mer is additionally tagged with the list of reference genomes in which it is contained.

Muggli et al. reduced the size of the colored de Bruijn graph in VARI [109] by replacing the hash map with BOSS [19] (a BWT-based [24] encoding of the de Bruijn graph 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 RRR [132] and Elias-Fano [43] encoding. Rainbowfish [5] 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 de Bruijn graph 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 de Bruijn graph 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 colored de Bruijn graph's associated color information.

In this paper, we develop a general scheme for efficient and scalable encoding of the color information in the colored de Bruijn graph 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 de Bruijn graph. We construct a weighted graph on the color classes in the colored de Bruijn graph, where the weight of each edge corresponds to the space required to store the delta between its endpoints. 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 [131] and the rooted hop-constrained MST problem [10]. Neither is in APX, i.e., it is not possible to approximate them to within any constant factor [102]. Althaus et al. gave an $O(\log n)$ approximation assuming the edge weights form a metric [10]. Khuller et al. 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 [72]. Marathe et al. show that in general we can find trees within $O(\log n)$ of the minimum diameter and weight [103]. 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 [123], a state-of-the-art large-scale sequence-search tool that uses a colored de Bruijn graph to index a set of sequencing samples, and the representation used in Rainbowfish [5], 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× and 2.5× storage savings relative to the representation previously used by these tools.

2 Method

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

2.1 Colored de Bruijn graphs

de Bruijn graphs are widely used to represent the topological structure of a set of k-mers [26, 55, 97, 122, 130, 144, 145, 166]. The de Bruijn graph induced by a set

of k-mers is defined below.

Definition 2.1. *Given a set E of k-mers, the* ***de Bruijn graph induced by E*** *has edge set E, where each k-mer (or edge) connects its two $(k-1)$-length substrings (or vertices).*

Colored de Bruijn Graphs extend the de Bruijn graph by assigning a ***color class*** $C(x)$ to each edge (or node) x of the de Bruijn graph. 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 [5, 7, 95, 109].
- Sometimes, U is a set of ***reads***, and $C(x)$ is the subset of reads containing x [1, 2, 158].
- Sometimes, U is a set of sequencing experiments, and $C(x)$ is the subset of sequencing experiments containing x [123, 149, 150, 153].

The goal of a colored de Bruijn graph representation is to store E and C as compactly as possible*, 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 $C(x)$.

Given that we can perform point queries, we can traverse the de Bruijn graph by simply querying for the 8 possible predecessor/successor edges of an edge. This enables us to implement more advanced algorithms, such as bubble calling [68].

*The nodes of the de Bruijn graph are typical stored implicitly, because the node set is simply a function of E.

Many colored de Bruijn graph representations typically decompose, at least logically, into two structures: one structure storing a de Bruijn graph and associating an ID with each k-mer, and one structure mapping these IDs to the actual color class [5, 109, 121]. The individual color classes can be represented as bit-vectors, lists, or via a hybrid scheme [164]. This information is typically compressed [118, 132, 168].

Our paper 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 de Bruijn graph structure representation of the counting quotient filter [121] as used in Mantis [123], our proposed color table representation can be paired with other de Bruijn graph representations.

2.2 A similarity-based colored de Bruijn graph representation

The key observation behind our compressed color-class representation is that the color classes of k-mers that are adjacent in the de Bruijn graph 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 repre-

sented 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 [17] in the context of information retrieval. Bookstein and Klein showed how to exploit the implicit clustering among bitmaps in 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 bitvectors 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 (see Figure 17). For each edge x of the de Bruijn graph, let $C(x)$ be the color class of x. Let $\mathcal{C}$ be the set of all color classes that occur in the de Bruijn graph. We first construct an undirected graph with vertex set $\mathcal{C}$ and edge set reflecting the adjacency relationship implied by the de Bruijn graph. 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 de Bruijn graph 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 de Bruijn graph. 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 endpoints (where we view color classes as bit vectors and $\emptyset$ is the all-zeros bit vector).

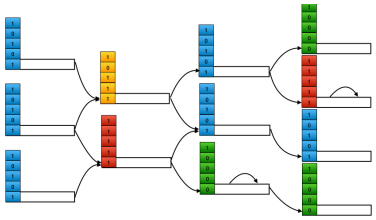
We then compute a minimum spanning tree of the color class graph and root the tree at the special $\emptyset$ node. Note that, because the $\emptyset$ node is connected to every other node in the graph, the graph is connected and hence an MST is guaranteed to exist. By using a minimum spanning tree, 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.

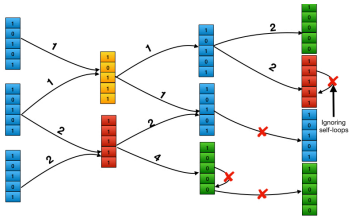
2.3 Implementation of the MST data structure

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:

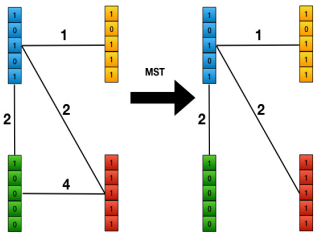
- **Parent vector**: This vector contains $|\mathcal{C}|$ slots, each of size $\lceil \log_2 \mathcal{C} \rceil$. The value stored in index i represents the parent color class ID of the color class with



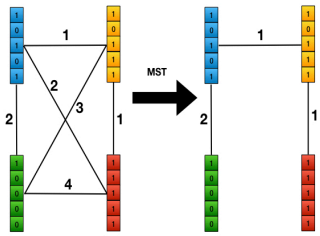
(a) A colored de Bruijn graph. Each rectangle node represents a k-mer. Each vector represents a color class (equal color classes have the same color).



(b) The color class graph from the colored de Bruijn graph. There is an edge between each pair of color classes that correspond to adjacent k-mers in colored de Bruijn graph. Weights on the edges represent the Hamming distances of the color class vectors.



(c) The color class graph we achieve from 17b by removing duplicate edges and its corresponding MST.



(d) The complete color class graph and its derived MST which has the minimum achievable total weight.

Figure 17: Encoding color classes by finding the MST of the color class graph, an undirected graph derived from colored de Bruijn graph. The order of the process is 17a, 17b, and 17c. The arrows in 17a and 17b show the direction of edges in the de Bruijn graph which is a directed graph. The optimal achievable MST is shown in 17d for comparison. Since we never observe the edge between any k-mers from color classes green and yellow in colored de Bruijn graph, 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.

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.

Query of the MST-based representation. Figure 18 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 [123] and Rainbowfish [5]. 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.

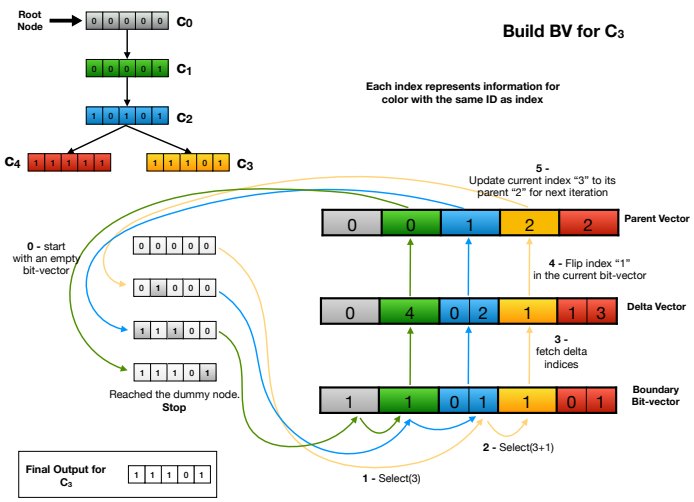


Figure 18: The conceptual MST (top-left), the data structure to store the color information in the format of an MST (right). This figure also illustrates the steps required to build one of the color vectors (C_3) at the leaf of the tree. Note that the query process shown here does not depict the caching policy we apply in practice.

For this paper, 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.

Construction. We construct our MST-based color-class representation as follows. First, we run Mantis to build its default representation of the colored de Bruijn graph. We then build the color-class graph by walking the de Bruijn graph and adding all the corresponding edges to the color-class graph. The edge set is typically much smaller than the de Bruijn graph (because many de Bruijn graph 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 in order 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" fashion. Specifically, Mantis stores the color class bit vectors on-disk sequentially by ID, grouped into blocks of roughly 6GBs 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.

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 endpoints (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, though many such optimizations are likely possible.

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 `bv` instead of the zero `bv` 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. Also, 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 more than 10 other queries and is more than 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 de Bruijn graph 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 de Bruijn graph?

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 range from 10^6 to 10^9, so the number of edges in the complete color-class graph would be on the order of 10^{12} to 10^{18}, or possibly even more, making this algorithm impractical for the largest data sets considered in this paper.

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 de Bruijn graph. 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 colored de Bruijn graph. 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 colored de Bruijn graph— to build our MST-based color-class representation?

- **Query performance.** How long does it takes to query the colored de Bruijn graph using our MST-based color-class representation?

3.1 Experimental procedure

System Specifications Mantis takes as input a collection of ***squeakr*** files [121]. 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 4-disk mirrors (8 disks total). Each is a Seagate 7200rpm 8TB disk (ST8000VN0022). They were formatted using ZFS and exported via NFS over a 10Gb link. We used different systems to run and evaluate time, memory, and disk requirements for the two steps of preprocessing and index building as was done by Pandey et al. [123].

For index building and query benchmarks, we ran all the experiments on the same system used in Mantis [123], an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB 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.

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 [121]. 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 dataset (2586 samples), the unfiltered Squeakr files required $\sim$ 2.5TB of space while the filtered files require only $\sim$ 108GB. 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 ***counting quotient filter*** (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 post-processing step on the standard Mantis color-class representation. For construction times, we re-

port only this post-processing step. This is because our MST-based color-class representation is a generic tool that can be applied to many colored de Bruijn graph 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(ENA) [114] in gzipped FASTQ format. Additionally, we have built the proposed index for 2,586 sequencing samples from human blood, brain, and breast tissues (BBB) originally used by [149] and also used in the subsequent work [150, 153, 164], including Mantis [123], 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 25.23TB.

In order 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 [149], by discarding k-mers that occur less than some threshold number of time. The thresholds are determined according to the size (in bytes) of the gzipped sample, and the thresholds are given in Table 15. We adopt a value of $k = 23$ for all experiments.

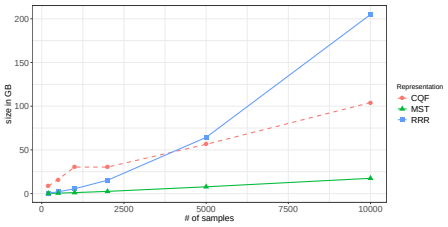
Min size	Max size	Cutoff	# of experiments with specified threshold
0	$\leq$ 300MB	1	2,784
$>$ 300MB	$\leq$ 500MB	3	798
$>$ 500MB	$\leq$ 1GB	10	1,258
$>$ 1GB	$\leq$ 3GB	20	2,296
$>$ 3GB	∞	50	2,864

Table 15: Minimum number of times a k-mer must appear in an experiment in order to be counted as abundantly represented in that experiment (taken from the SBT paper). Note, the k-mers with count of "cutoff" *are* included at each threshold.

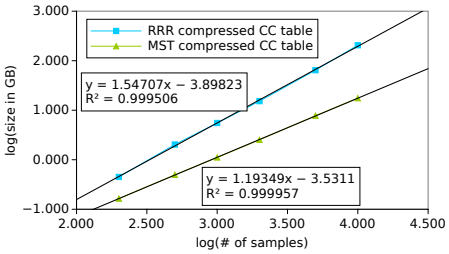
3.2 Evaluation results

Scalability of the new color class representation Figure 19a and Table 16 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 19a also plots the size of the CQF that Mantis uses to map k-mers to color class IDs. We can draw several conclusions from this 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.



(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 $\approx \Theta(n^{1.2})$.

Figure 19: Size of the MST-based color-class representation vs. the RRR-based color-class representation.

Table 16 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 dominate the MST representation both in terms of total size and in terms of growth. Table 16 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 de Bruijn graph-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 colored de Bruijn graph representation, we plot the sizes of the RRR-based color-class representations and MST-based representations on a log-log scale in Figure 19b. Based on the data, the RRR-based representation appears to grow in size at a rate of

roughly $\Theta(n^{1.5})$, 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 16 show the size of the RRR- and MST-based color-class representations for the human blood, brain, breast (BBB) and *E. coli* data sets respectively. BBB is the data set used in SBT and its subsequent tools [150, 153, 164], as well as in Mantis [123] and *E. coli* is the data set analyzed in the Rainbowfish paper. This dataset, which has been obtained from GenBank [116], consists of $5,598$ 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 dataset since Rainbowfish has already demonstrated a large improvement in size for it compared to Vari [109].

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.

Index Building Evaluation The "Build time" column in Table 16 shows the time required to build our MST-based color-class representation from Mantis' RRR-based representation. All builds used 16 threads. Table 18 shows how the MST construction time for a 1000 sample dataset 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

Dataset	# samples	RRR matrix	MST						$\frac{size(MST)}{size(RRR)}$
			Total space	Parent vector	Delta vector	Boundary bit-vector	Build time (hh:mm:ss)	Expected edge weight	
H. 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
	1,000	5.14	1.03	0.37	0.6	0.06	0:25:03	4.39	0.2
	2,000	14.2	2.35	0.71	1.5	0.14	0:51:58	5.38	0.17
	5,000	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
Blood, Brain, Breast (BBB)	2586	15.8	2.66	0.63	1.88	0.16	00:57:43	6.98	0.17
E. coli strain reference genomes	5,598	2.06	0.83	0.02	0.76	0.06	00:03:15	7.8	0.4

Table 16: Space required for RRR and MST-based color class encodings over different numbers of samples (sizes in GB) and time and memory required to build MST. Central columns break down the size of individual MST components.

Dataset	# samples	Mantis Build memory (GB)	MST Build memory (GB)
H. sapiens RNA-seq samples	200	5	8
	500	10	16
	1,000	18	29
	2,000	25	29
	5,000	58	59
	10,000	111	111
Blood, Brain, Breast (BBB)	2586	28	29

Table 17: The memory required for Mantis build and MST compression phases on human RNA-seq data. 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.

and the MST construction steps is shown in Table 17. 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.

# 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

Table 18: The MST construction time for 1000 experiments using different number of threads. Memory stays the same across all the runs.

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 one week. Thus MST construction represents less than 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 $111G$.

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.

	Mantis			Mantis		
	index load + query	query	space(GB)	index load + query	query	space(GB)
10 Transcripts	1 min 10 sec	0.3 sec	118	32 min 59 sec	0.5 sec	290
100 Transcripts	1 min 17 sec	8 sec	119	34 min 33 sec	11 sec	290
1000 Transcripts	2 min 29 sec	79 sec	120	46 min 4 sec	80 sec	290

Table 19: Query time and resident memory for mantis using the MST-based representation for color information and the original mantis (using RRR-compressed color classes) over $10,000$ experiments. 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.

Table 19 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 sec / 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 & Conclusion

We have introduced a novel exact representation of the color information associated with the colored de Bruijn graph. Our representation yields large improvements in terms of representation size when 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 ~ 0.08 sec / query. Further, 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 colored de Bruijn graph'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 de Bruijn graph.

Though 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. [163], 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 counting quotient filter, which is used to store the topology of the de Bruijn graph and to associate color class labels with each k-mer, has become the new scalability bottleneck. Here, 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 de Bruijn graph share the same color class ID, it is likely possible to encode this label information sparsely across the de Bruijn graph, taking advantage of the coherence between topologically nearby k-mers. Further, to allow scalability to truly-massive datasets, 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 de Bruijn graph. Nonetheless, because we have designed our color class representation as essentially orthogonal to the de Bruijn graph representation, we anticipate that we can easily integrate this approach with improved representations of the de Bruijn graph.

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

Chapter 6: An incrementally-updatable and scalable system for large-scale sequence search using LSM-trees*

1 Introduction

The databases that house large public collections of sequencing experiments have been growing exponentially throughout the past decade, owing to cost-efficient and accessible high-throughput sequencing technologies. Many of these experiments are publicly available and widely accessed — for example, in the Sequence Read Archive (SRA) [76, 81]. The experiments contained in such databases range across different species, including human, bacteria, viruses and plants, and are associated to a variety of studies, such as tracking the effect of a genetic disease, drug treatment, or the environment on the studied genome or transcriptome. Considering the meta-information coming with the reads, such a database is a rich resource for exploratory analyses and novel sequence-level discoveries to answer biological questions. For instance, having a newly-assembled transcript or gene, a question of interest is "what are the list of experiments that contain this sequence?" Such a question can be classified as a search problem. Specifically, this is the problem of searching for a sequence, across a large database, for experiments that contain identical or highly-

*A collaboration with members of the Mantis team

similar sequences to the query sequence. The problem of finding similar sequences has been tackled in the field of sequence alignment which is a well-studied subject with many popular solutions, indices, and tools available [6, 40, 78, 85, 86]. In fact, through BLAST [11] and its variants, one can access and search (either through the web interface or in a locally-installed database) the vast catalog of assembled genomes.

However, the problem of searching for a sequence in a database of raw, unassembled sequencing reads is fundamentally different from the problem of searching through a database of assembled genomes. One main difference is that the sequenced reads themselves are highly fragmented and repetitive compared to long assembled genomes and transcripts that usually act as the reference in typical formulations of the sequence alignment algorithm problem. This renders an attempt at calculating the edit distance of two sequences using dynamic programming (even with efficient heuristics), essentially ineffective as a long query sequence may not be present in its entirety in an experiment. Furthermore, the size of the database of raw data is much larger than that of the assembled sequences for many reasons including, the fact that many more individuals have been sequenced than those whose genomes have been assembled, as well as the fact that even the same individual (or cell line) may have its dynamic transcriptome sequenced many times under different conditions or stimuli. Even more, since the vast majority of the diverse life on earth cannot be reliably cultivated in a laboratory, we have evidence of the genomes of such organisms only though shotgun metagenomic sequencing, resulting in frag-

mented and incomplete coverage of the underlying genomes; though such data is not presented to us in the ideal form, it is, nonetheless, valuable data.

As a result of the collection of such data, the SRA currently has $\sim$ 4 petabases of publicly-available sequencing data. This has led to the proposal, over the past few years, of new algorithms and data structures for indexing these large databases of short read sequencing experiments. This problem was first introduced by Solomon and Kingsford [149], and they suggested the Sequence Bloom Tree (SBT) as a data structure for indexing such sequencing databases for "experiment discovery". Since this original work, many tools, algorithms, and data structures have been proposed to index large databases of short-read sequences [8, 15, 21, 59, 124, 150, 153, 165]. Almost all of these approaches start with breaking the sequences in the given sample into sub-sequences of size k, known as k-mers, and defining the intersection of the query k-mer set and each sample's k-mer set as the criterion for determining the likelihood of query presence in the sample. Among the tools proposed in this space, only Mantis is exact [124]. Most of the other indices that have been designed specifically for large-scale sequence search yield approximate search results. That is, all samples that truly meet the intersection criterion will be returned, but so will some (theoretically-controllable) fraction of samples that do not satisfy this search criterion. This behavior of reporting a query as present at the requisite level, while, in reality, it is not, is analogous to a false-positive results.

As explained by Harris and Medvedev [59], raw sequence indices can be divided into two main categories: those that aggregate k-mers at the level of experiments

(experiment-based) and those that aggregate experiments at the level of k-mers (k-mer-based). In the first category (experiment aggregation), there is usually a separate structure for each experiment such as a small filter for the k-mers in the experiment (e.g. Bloom filters). These data structures are then aggregated either in a hierarchical manner [59, 149, 150, 153, 165] or in a flat fashion [15, 21] into one, unified index. In the second approach (k-mer aggregation), unique k-mers across all experiments are aggregated into one structure which maps the k-mer to another structure containing a list per k-mer which indicates its presence/absence in each experiment (called the "color" of the k-mer) [110, 112, 124]. In the later design, the challenges of identifying k-mers and the that of storing the appropriate colors, can be looked as two fundamentally different problems, and the approaches to tackle these problems can be developed and improved independently. On the other hand, in the experiment aggregation approach, since there is a separate structure for each experiment, the index can be loaded into memory in parts, which maintains a low query memory. For the same reason, updating the index (e.g. adding a new experiment) is a more straightforward task in the experiment aggregation approaches.

For example Mantis, an index from the k-mer aggregation category, is up to two orders of magnitude faster to query than state-of-the-art experiment aggregation-based approaches. However, given that it combines the k-mers of all experiments into a single structure to associate metadata with each that spans across all experiments, is not immediately clear how an index such as that implemented in Mantis can be updated. On the other hand, updates in experiment aggregation-based tools such as SBT and its SBT-variants [59, 149, 150, 153] are conceptually more clear. However,

though these approaches admit a more theoretically straightforward update procedure, to our knowledge, only the original SBT tool (the simplest of the SBT-based variants) actually implements such updates. On the other hand, Pandey et al. [124] describe how a potential update scheme may be applied to the Mantis structure by integrating the index into a multi-tier data structure that facilitates low-cost insertions, namely, making use of log-structured merge trees (LSM-trees) [120, 136, 137].

In this work, we put the original suggestion of [124] into practice, while simultaneously incorporating the subsequent improvements that have been made to the Mantis data structure since the original work (that make the data structure more efficient, but also more complex). "Log-structured Merge" is a general and well-understood technique used in various storage schemes to support fast insertion of stream of data [120]. The main operation that an index must support to allow for the use of an LSM-tree is merging — building a new index from two smaller ones without the need to reconstruct the whole index from scratch. Accordingly, we provide an algorithm for merging two Mantises into one larger index, and tackle the main obstacles of scalability and efficiency along the way. We note that the idea of enabling a raw sequence index to scale to larger collections of data via merging is not new to the field, and has been recently explored in previous work such as Varimerge [112]. The focus of this work, however, is to design a merging methodology specialized for Mantis data structures. We present efficient merging algorithms for both Classic Mantis and MST-based Mantis. In Classic Mantis, although the merging algorithm may at first seem simple, a naïve implementation is very resource

intensive. We use insights about the empirical distributions of data and careful engineering to make the merging memory scale moderately with the growth of the index. Specifically, we break the singular k-mer "map" (implemented via the counting quotient filter (CQF) [121] in Mantis) into smaller partitions based on k-mer minimizer values. This idea has been comprehensively explained and used previously in BCALM2 [31] for extracting non-branching paths of the de Bruijn graph over a set of raw experiments without the need to construct the full de Bruijn graph itself. For the MST-based Mantis data structure, we go further to design an efficient algorithm for merging two MSTs, which is the encoding used in MST-based Mantis to compress color information. This vastly reduces the intermediate disk space as well as improves the total merging time. Furthermore, with careful engineering to make efficient use of multi-threading, we retain a time-efficient construction process. Comparing the merging benchmarks with Varimerge, a recent state-of-the-art colored de Bruijn graph representation utilizing a merge-based construction, we observe better performance in merge time, merge memory, intermediate disk space, and final index size. We also benchmark the query time in this new index that makes use of the partitioned k-mer map structure with MST-based Mantis (one of the fastest existing raw sequence search indices). We observe similar total query time while obtaining up to a $7x$ improvement in memory for the largest indices.

2 Method

There are a number of steps required to enable a dynamically-updatable Mantis index that is scalable to large collections of data. As explained in 1, the way we achieve this goal is by use of LSM-trees. Figure 20 shows how an LSM-tree of multiple Mantises operate. The update process is comprised of two main steps: insertion, and compaction. Every time we add a new experiment of raw sequences to the database, we call `insert` which adds the sample to the list in RAM in the form of a squeakr [121]. A squeakr is basically a multiset representation holding the list of distinct k-mers in the experiment. If the first level in the LSM-tree is full, we flush the Squeakrinto a Mantis index and `merge` it into the first level on disk (level 1) in the `compaction`. This could trigger a cascading merge of the indices to higher levels until the destination level is no longer full. For this design to work, the one primary requirement to satisfy is that the `merge` operation be support by the Mantis indices atop which this structure is built. Therefore, the rest of this section covers to the details of the process that has been developed to enable an efficient and scalable merging operation for the Mantis index.

A Mantis index [124] is essentially a colored de Bruijn graph (colored de Bruijn graph) representation. In a colored de Bruijn graph, each k-mer has an associated color, which lists the experiments containing that k-mer. Mantis uses counting quotient filter [121], a key-value filter, combined with a color-encoding scheme adopted from [4] to map each k-mer to the color bit-vector of size equal to number of experiments encoding the presence/absence of the k-mer in an experiment. Subse-

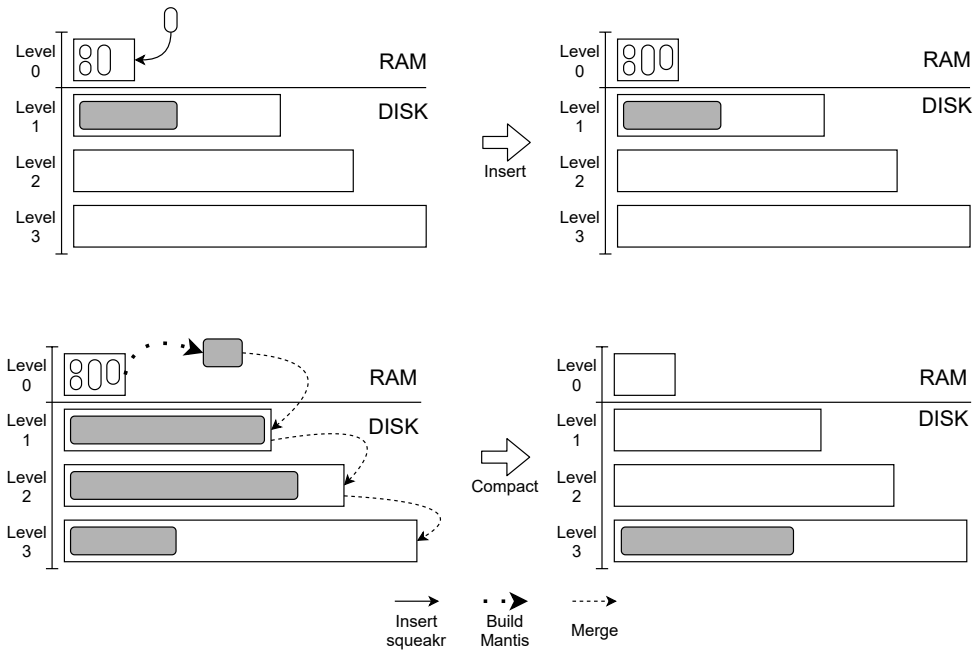


Figure 20: The two main steps during an LSM-tree update with slight modifications specializing it for the Mantis index. In *insertion* step, we only add a new Squeakr to the list of Squeakrin memory. In *compaction* step, which is triggered whenever number of Squeakrpasses the threshold for level 0, we first build a Mantis index over all the Squeakrin level 0 and then do a cascading merge of Mantis indices up to the point that no levels are full anymore. We should note that the levels are not drawn to scale for having a better graphical representation.

quently, the MST-based Mantis index [9] was introduced as the successor of Mantis. It focuses primarily on reducing the index size by performing referential encoding of related color vectors, rather than requiring identical k-mer occurrence vectors. This allows an even smaller index that can be effectively scaled larger even larger databases.

In section 2.1, we explain the merging process over "Classic Mantis", which although algorithmically simple, is quite resource intensive. We further explore the particular challenge we face for having a memory-scalable merge in section 2.3. This challenge, in particular, is representing all unique k-mers in a single counting

quotient filter that grows linearly with the number of unique k-mers. The existing Mantis indices (both Classic Mantis and MST-based Mantis) require this monolithic counting quotient filter to be present in memory during both construction and query. To replace this component of the index, we propose a data-driven partitioning scheme based on minimizers that breaks the single counting quotient filter into smaller partitions that can be loaded in memory and processed independent of other partitions during construction, merge, and also query time. We then explain how this greatly improves the memory required for counting quotient filter merging. We then describe an efficient algorithm for merging two MSTs in section 2.2. Direct merge of the MSTs allows us to skip the construction and storage of the very large intermediate color class matrix to disk and to avoid the extra step of compressing this matrix into the final MST representation. This helps reduce the intermediate disk space as well as improves the total time required for merge. Finally, we explain how we design the construction and merge procedures to be parallelizable allowing for practically fast construction.

2.1 Merging Classic Mantis indices

The Mantis merge algorithm takes $cdbg_1$ and $cdbg_2$ as inputs, and produces an output Mantis index over all the experiments of the union of the input indices. The merge algorithm can be sequenced into three major steps: (1) gathering all the distinct color-id pairs for the union set of k-mers of the input graphs, where each color-id pair represents a distinct color for the k-mers of the output graph; (2)

assigning color-ids to these pairs and building the color-class table for these colors and (3) building the merged de Bruijn graph, (i.e. the output counting quotient filter containing the union k-mer set). Note that the sets of experiments for the two input indices should be disjoint; otherwise each copy of an experiment will be treated as a different experiment at the merged index. We describe these steps below.

Gathering distinct color-id pairs. Each k-mer *key* in a Mantis index has an associated non-zero color-ID *id*. If some k-mer *key* has color-ID id_1 at the first input index $cdbg_1$ and color-ID id_2 at the second input index $cdbg_2$, then this k-mer *key* will be assigned a unique color-ID *newid* corresponding to the pair (id_1, id_2) at the merged index. It is possible that the k-mer is absent in either of the indices. In such a case, the color-ID value of the k-mer in the input index which it is not present is considered to be zero, i.e. $id_1 = \mathbf{0}$ or $id_2 = \mathbf{0}$. At the end of this phase, we will have all the distinct (id_1, id_2) color-ID pairs for the union k-mer set of the two input indices.

The color class matrix is partitioned and stored as fixed-size blocks of color classes to the disk, instead of storing the table as a whole. This facilitates in keeping the working memory low, as a color class is not required to be present during the full lifetime of the Mantis merge algorithm. Let the input indices $cdbg_1$ and $cdbg_2$ have partitions d_1 and d_2 in the color class table. For simplicity, we assume color class partitions to be $1-based$ indexed. We initialize $(d_1+1) \times (d_2+1)$ files on disk (called disk-buckets) to temporarily store color-ID pairs of k-mers. A disk-bucket

$b_{i,j}$ with $i, j > 0$ contains the color-ID pairs (id_1, id_2) where id_1 belongs to color class partition i and id_2 belongs to color class partition j. In the case the k-mer is not present in one of the inputs, the color-ID pair would end up in a disk-bucket with $i = 0$ or $j = 0$. Disk-bucket $b_{0,0}$ is empty as it is, by definition, associated to the color-ID pairs where the associated k-mer is neither present in $cdbg_1$, nor in $cdbg_2$.

Since the counting quotient filters of the indices $cdbg_1$ and $cdbg_2$ contain the hash-values of the k-mers in sorted order, we make simultaneous linear scans over the two counting quotient filters for the distinct k-mers of the union counting quotient filter. For each distinct k-mer key of the union counting quotient filter, let its color-IDs be id_1 and id_2 from $cdbg_1$ and $cdbg_2$. If id_1's color class is in partition x and id_2's color class is in partition y, then we store the pair (id_1, id_2) to bucket $b_{x,y}$, with possible repetitions of the pairs. Having completed the union process, we filter out the unique k-mers at each bucket (by sorting and discarding duplicates of the k-mer hashes in the buckets).

Simply following this scheme to collect the distinct pairs results in too much repetition of pairs and, thus, a high memory demand. Since we store a color-pair per k-mer, the disk-buckets will cumulatively contain exactly n_k pairs if the merged colored de Bruijn graph has n_k k-mers, whereas the number of color classes is actually much smaller. In reality, not only can multiple k-mers can share the same color, but also the majority of k-mers actually belong to a small subset of colors, less than 1% of total colors [124] in a typical dataset. The original Mantis data structure leverages this highly skewed abundance distribution of color classes by sorting the color classes based on their abundance values, and assigning IDs in reverse order

of the abundance — *at least* for the colors observed in a sample subset. Since in the final space required to store each color-ID correlates with the ID value itself, assigning smaller IDs to more abundant colors results in smaller size representation of the color-IDs. This utilization of the highly-skewed color distribution is popular in many other tools in the field as well [4, 6, 62]. We perform a sampling phase in which we analyze the color class abundance distribution of a subset of the k-mers. We sort the color class based on their abundances in the sample set, and keep a fixed number of the (approximately) most abundant pairs in a hash-map H which maps a color-ID pair to its abundance. Given the uniform-randomness property of the hash function used to hash k-mers, in expectation, we will see the most abundant color classes in the first few million k-mers. After this sampling phase, we make the simultaneous linear scan over the counting quotient filters and for each color-ID pair (id_1, id_2) of a k-mer, we only add the pair to its corresponding bucket if it is not present at H.

Assignment of color-IDs and building the color class table. At this point of the algorithm, we have all the distinct color-ID pairs for the output Mantis index in the hash-set H and in the disk-buckets. We assign a unique integer as the color-ID of each pair and then build the output color class table. As explained in the previous paragraph, we assign IDs to the colors in the hash-set H in reverse order of the abundances. Then we assign color-IDs to the rest of the pairs present at the disk-buckets. Later, throughout the process, we would need to have a constant access from each input color-ID pair to its associated output color-ID. However, using a

naive hash-map on all the non-sampled color pairs as keys is very memory-inefficient. Instead, we use a space-efficient hashing scheme, in our case the Minimal Perfect Hash Function(MPHF) implemented by Limasset et al. [93]. Knowing the count of color pairs in each bucket, we build a MPH table $MPH_{i,j}$ on it. That is, if the bucket $b_{i,j}$ has s pairs, then each pair from $b_{i,j}$ gets mapped to a unique value in the range $[0, s-1]$ by $MPH_{i,j}$. Then the color-ID of a pair (id_1, id_2) present at bucket $b_{i,j}$ is computed as the sum of the following: count of abundance-sampled pairs (i.e. $|H|$), cumulative size of all the buckets up-to this bucket (exclusive and in row-major order), and the pair's MPH value. This scheme ensures a unique color-ID for each pair present at each disk-bucket.

After the color-ID assignment, we sort the color-ID pairs (id_1, id_2) present at H in such a way that all the pairs that have id_1's color class at some partition x of $cdbg_1$ and id_2's color class at some partition y get grouped together. Then we scan over this sorted order, load the color class table partitions of the input indices only when required (i.e. we switch from one group to other), and build the color classes for the pairs. Finishing H, we go over each disk-bucket $b_{i,j}$, load the color classes table partition $(i-1)$ of $cdbg_1$ (if $i > 0$) and partition $(j-1)$ of $cdbg_2$ (if $j > 0$), and build the color classes for each pair present at $b_{i,j}$, in order of their newly assigned color-IDs. Also, partitions of the output color class table are serialized to disk once they reach a certain threshold, as was originally described in [124]. This color class building scheme for H and the $b_{i,j}$'s ensures that we only need to load two color class buckets at a time in memory.

Building the merged counting quotient filter At this point of the algorithm, we have all the distinct color-ID pairs for the union k-mer set and their associated color-ID. Similar to the first phase, we make simultaneous linear scans over the input counting quotient filters, this time for each distinct k-mer in the union of the two inputs, storing the $\langle < k-mer, colorID \rangle$ pair to the output counting quotient filter. For each distinct k-mer key, we get its color-ID pair (id_1, id_2), query for the pair's assigned color-ID in the hash-map H, and if absent, compute its color-ID from its corresponding MPH table as discussed earlier.

2.2 Merging MSTs

A Classic Mantis index over n samples (also considered a colored de Bruijn graph representation), consist of two main sub-structures, a counting quotient filter which maps each distinct k-mer to a color-ID and a color class matrix that maps the color-ID to a color bit-vector. One can then construct an MST-based Mantis index from a Classic Mantis index by compressing the color class matrix into a relative encoding of the color bit-vectors as described in [9]. The idea is to exploit the inherent similarity of the rows in the color class matrix. Rather than directly storing the full matrix on disk, each color bit-vector (a row in the matrix) is encoded as a list of deltas from neighboring color bit-vector. Applying this idea globally results in a tree structure (a MST) over a derived color graph from Classic Mantis that connects similar color bit-vectors. In the rest of the section we call this encoding a MST for short. When applied, this compression technique significantly reduces the space

occupied by the color information and is progressively more effective with increasing numbers of samples to the extent that in previous experiments it achieves more than 80% saving on the color information and close to 50% global saving compared to Classic Mantis for an index on $10k$ samples.

At query time, for each color-ID c_i, the color bit-vector needs to be reconstructed by traversing the path node color-ID c_i up to the root and flipping the bits whose positions are encoded by the deltas observed along the traversed path. Although this process is not a constant-time observation, it has been shown that using a small LRU cache (Least-Recently-Used cache) to explicitly represent a dynamic subset of nodes of the MST yields practically the same query time as direct access to the color-vector in Classic Mantis. The color classes in the LRU cache get updated based on the frequency of access request to the color classes either by direct query for the corresponding color-ID or along the path to the root for other color-IDs.

To avoid lengthy (re-)construction of the MST representation from a color class matrix after every merge of two Classic Mantis indices, it would be highly-preferable to directly merge MSTs instead. Not only does this save time, but it eliminates the large disk-space requirements of the color class matrix. The process we propose for achieving this goal is similar to that of the MST construction from the color class matrix itself, but with appropriate modifications to account for lack of direct access to color bit-vectors.

Therefore, we first provide a summary of the MST construction process as explained in [9], and then go over the modifications made to enable direct construction of the final MST from input MSTs.

The following are the main steps of MST construction process:

1. Construct a "color-graph"($\langle V, E \rangle$) where V is the set of all color-IDs in the colored de Bruijn graph and E filled by walking the colored de Bruijn graph and storing an edge between the colors of the neighboring k-mers in the colored de Bruijn graph if they have distinct color-IDs (self-loops are not allowed in the color-graph).

2. Creating an edge from all color nodes to a dummy node. The dummy node represents an empty `bv` with all bits reset to 0. Assuming the colored de Bruijn graph has n distinct color classes, the color-graph will contain $n + 1$ nodes (including the dummy `bv` node). This guarantees the color graph will be one connected component.

3. Calculate the weight of each edge of the color-graph as shown in part (a) of 21. The weight of an edge $\langle c_i, c_j \rangle$ is defined as the hamming distance between the color bit-vectors associated to c_i and c_j (exclusive-or of the two bit-vectors as shown in part (b) of figure 21). At this point, each color bit-vector C_i in the color-graph has at least one edge to another color bit-vector C_j where the edge weight represents number of indices required to be stored to retrieve color bit-vector C_i from C_j if we already have the color bit-vector of C_j.

4. Find the MST of the color-graph, which is the tree with minimum total weight that spans all the color-IDs. Therefore, the vertices of the color-graph remain connected, while the weight of the resulting spanning tree (which allows reconstruction of each color bit-vector) is minimised.

5. Knowing the root of the tree (the dummy node), orient the edges of the tree by walking the tree and assigning the parent-child directions to each edge.

6. For each edge $\langle c_i, c_j \rangle$ in the MST, calculate the offsets of the unequal bits between the color bit-vectors associated to color-IDs c_i and c_j.

7. Store the tree representation as the "Parent vector" and "Delta vector". The parent vector stores the structure of the tree by pointing each color-ID to its parent color-ID and the delta vector encodes the list of indices for the nonidentical bits between the color-ID and its parent's corresponding color bit-vector.

The MST construction steps (subsequent to color-graph construction) are also depicted in 21, part (a). It is important to note that the first step of merging two colored de Bruijn graphs, i.e., the counting quotient filter merge, remains the same in both Classic Mantis and MST-based Mantis. The two outputs of the counting quotient filter merge step are "the output counting quotient filter" and "the color-map". The counting quotient filter maps each unique k-mer in the union of the left and right samples to its associated color-ID, and the color-map maps each color-ID to the pair of corresponding color-IDs in the left and right input index $c \rightarrow \langle c_l, c_r \rangle$. Therefore, the available structures before starting the MST construction are the merged counting quotient filter, the color-map connecting the new color-IDs to pair of colors for the left and right indices ($c \rightarrow \langle c_l, c_r \rangle$) and the two MSTs of the left and right indices.

The majority of the steps in MST merge are the same as the ones in MST

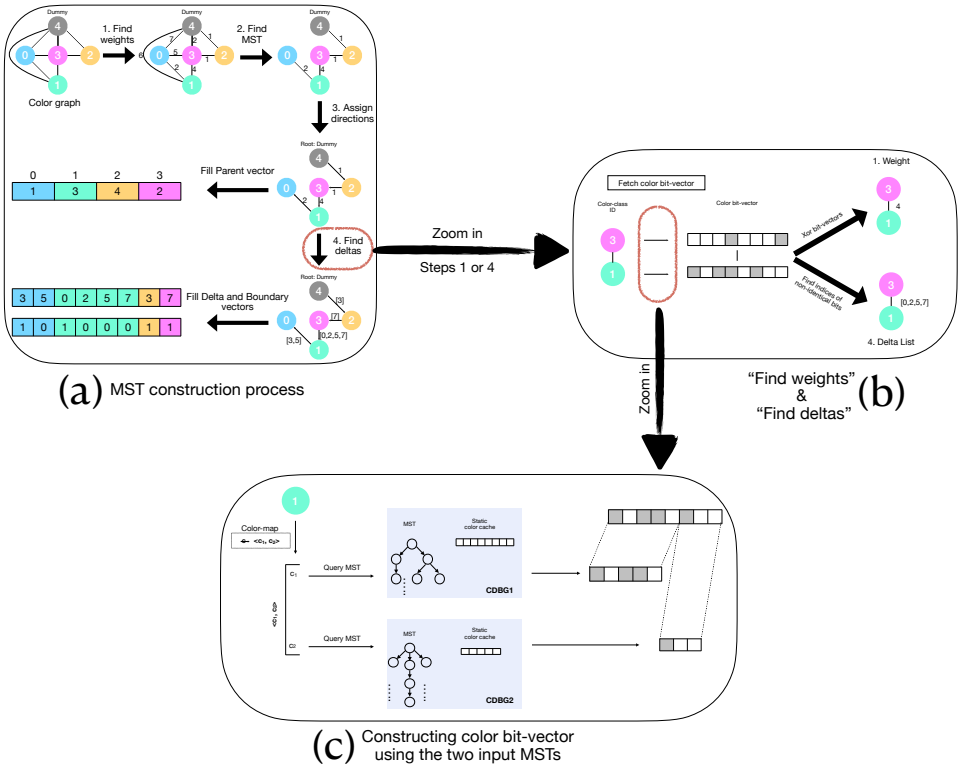


Figure 21: The full MST construction process (a) which is zoomed in two levels down in (b) and (c). The process in (a) basically follows the same steps as for MST construction in the MST-based Mantis index. The main point of difference is reconstruction of the color bit-vectors by querying the two input MSTs as shown in (c) rather than fetching the vector from the color class matrix. During the MST construction, step (c) will be repeated for each end of the edge in (b) and eventually for all the edges in the original color-graph (step 1) and later the final MST (step 4) in (a). Consequently, the main challenge is to make the color bit-vector reconstruction process as efficient as directly querying the color class matrix for the corresponding row (i.e., constant time).

construction. In the next few paragraphs We discuss the pipeline for MST construction/merge describing individual steps in figure 21.a and point out the specific differences between color class matrix merge and MST merge. The color-graph populating step is exactly the same when constructing an MST or merging two MSTs. We walk the counting quotient filter in order of the k-mer hash values and for each pair of $\langle kmer_i, colorID_i \rangle$, we find all neighboring pairs $\langle kmer_j, colorID_j \rangle$ and add the color-ID pair $\langle colorID_i, colorID_j \rangle$ to the edge list if $colorID_i \neq colorID_j$.

For step 1, finding the weight of the color-graph edges, we need to access the

color bit-vectors associated to each end of the edge. If we have the full color class matrix available, this only requires a (constant-time) lookup to fetch the corresponding row in the matrix that a color-ID references. However, this step is not trivial when we only have access to the two input MSTs. We know from section 2.1 that, in the merging process, each color bit-vector in the output is a concatenation of some color bit-vector from the left and right inputs. We store the mapping of the output color bit-vector IDs to the ID pair of the two constituent input color bit-vectors in a map called color-map. So, for each color-ID c_i that maps to two input color-IDs (i.e. $c_i \rightarrow \langle c_m^l, c_n^r \rangle$), if we are able to construct the color bit-vector associated to c_m^l in the left index, cbv_m^l, and c_n^r in the right index, cbv_n^r, we can construct the cbv_i^o by concatenating them ($cbv_i^o = cbv_m^l | cbv_n^r$). In the case of in which we have MST-based Mantis indices as the left and right input, to retrieve the color bit-vector associated to a color-ID, we need to query the MST of each of these indices (the process for which is explained in details in [9]). To summarize, to calculate the weight of an edge $\langle c_i, c_j \rangle$, we first extract the color bit-vectors for the associated id pair to c_i, $c_i \rightarrow \langle c_m^l, c_n^r \rangle$, by querying the left and right input MSTs. Then we concatenate the two vectors to make the color bit-vector cbv_i. We repeat the same process for node c_j to construct cbv_j. Then the weight of the edge would be the Hamming distance between the two vectors (cbv_i, cbv_j). We need to repeat the same process for all edges in color-graph.

At this step, however, we encounter a challenge. Although, the LRU cache was sufficient to enable querying speed in MST-based Mantis that is practically equivalent to the constant-time access to the color class matrix, we do not observe

such behavior during merging of the input MSTs. In this case, the query pattern that is dictated by the color-graph is fundamentally different in its distribution compared to (later) querying the index with specific query sequences. Thus, the LRU cache is insufficient to enable efficient use of the input MSTs as color vector representations during the merge process. Instead, we design a static caching scheme that is optimized to make the queries we require practically fast. This scheme is described in subsection 2.2.1.

Step 2 is a straightforward and purely algorithmic process of finding the MST of a weighted graph, i.e. the color-graph. At step 3, we assign the directions of the edges in the MST to be able to traverse the tree (as a requirement for the final encoding representation of the colors). We start a BFS (Breathe-First-Search) or DFS (Depth-First-Search) walk from the root of the tree which is known (dummy node) and fill the parent vector accordingly. The parent vector has $\|V\|$ slots, equal to number of nodes in the tree. For each $c_i \in children(c_p)$ in the MST, we set the value at index c_i in the parent vector to c_p. To complete the output MST construction, we need to store the delta indices between the color bit-vectors for each parent-child pair in the last step, step 4. This process also requires, for each edge of $c_p \rightarrow c_c$ in the MST, to construct the color bit-vectors of the two nodes c_p and c_c and store the indices of the non-identical bits. The color bit-vector construction follows the exact same protocol as followed for calculating edge weights. The associated color bit-vectors for the left and right input color-IDs are fetched and and concatenated to build the output color bit-vector. Since the list of delta indices varies across the edges, we use another bit-vector to denote the start index for each

delta set. At this point, we have constructed the output MST without having had to to construct the intermediate color class matrix at all.

2.2.1 Static cache

As explained in [9], to query a color-ID from an MST, we need to walk a path on the MST from the color-ID node to the root to assemble the color bit-vector applying the relative bit differences between the parent and child color-IDs along the path. This will lead to a color-ID query time for MSTs that scales with the height of the MST, and would in an unpredictable way with the number of color classes. The idea proposed to practically overcome this issue was to adopt an LRU cache (Least-Recently-Used cache) to always keep a subset of the fully decoded color classes in memory. The policy for populating this LRU cache was based on the query popularity of color classes, as well as on how many times internal nodes of the tree are visited during the query procedure. Queried color-IDs were selected and added to the LRU cache if not already there, as were nodes along the path from the queried color-IDs to the root depending on the number of times they'd been traversed. If the LRU cache is full, the color class that has not been requested for the longest amount of time would be evicted from the cache. Practically, the more a color-ID is requested, the higher the probability that it would be found in the LRU cache. This dynamic cache worked well to accelerate query in the MST.

Unfortunately, in the case of querying an input MST in the MST merging process, the the LRU cache is not very effective. This is primarily the case because,

during the MST merging process, many pairs of input color-IDs are accessed, and the probability of accessing a color-ID is unrelated (at least in any obvious way) to it's popularity among k-mers. Specifically, during merge, every single color-ID from each input must be queried at least once, because otherwise, a color class pattern would be missing in the merged output. This immediately changes the scope of the problem from, for instance querying only 10% of the color classes 90% of the time, to querying 100% of color classes, regardless of their frequency. Also, each color-ID will be queried whenever it appears as one end of an edge. Assuming we have n color classes in our color-graph, each color-ID c_i^l can be paired with up to $n-1$ other color-IDs in the color-graph. This inherent multiplicity further reduces the efficacy of an LRU-caching scheme. Even if one sorted the edges, the issue of multiplicity still remains. For example, consider the following case; we have an LRU cache of size L for our MST and each color class is connected to L other color classes where $L << n$ total number of color classes. Assume the best-case scenario where the edges are sorted based on source and destination color-IDs. When we observe an edge $\langle c_i, c_j \rangle$, we insert both color classes into the LRU cache. For the next $L-1$ edges, we would have the c_i in the cache an not require to walk the tree, but, we query all the destination color-IDs and also add them to the LRU cache. After L edges starting with c_i, we would go to the edges with a different source node c_n (lets say starting from edge $\langle c_n, c_j \rangle$), although we have already observed c_j but right before the current edge, c_j has been kicked out of the cache as the longest recently used color-ID. So, the nodes we are searching for would not be in the LRU cache, and we would need to reconstruct c_j again. This would occur as many times as we

observe c_j as the destination node of some edge, so that in the worst case we need to query MST for c_j for all the L times that it occurs.

To manage the MST query time without the need to keep all the color bit-vectors in memory, we must design a cache that is more aware of the characteristics of the queries. One important advantage of querying the MST during merging compared to regular sequence queries, is that the color-IDs to be queried are known ahead of time. Since we have list of the edges (as pairs of color-IDs), we know exactly what color-ID pairs appear and how many times each color-ID would be queried. We use this information to design a "static cache" that is filled once before the start of the query step and then used during the entire MST merge process. It can be tuned for the lookup cost that the user wants to achieve, trading off memory for more efficient lookup.

In this cache, we define "cost" as the number of steps required to get from the queried color-ID to the root, including the color-ID itself. We then design the cache as follows: We consider two types of cost for each node in the tree; (1) the local cost is the total number of times the node is directly queried and (2) the sub-tree cost is the total cost of the sub-tree rooted at the current node. We also define the parameter c as the average cost per query. This is average number of steps required to walk in the MST tree before either (1) reaching the root or (2) hitting a node whose corresponding color bit-vector exists in the "static cache".

Now we can fill the "static cache" so that we guarantee the average cost of c per query. The cache is constructed as follows:

1. Walk over the edges and calculate the local cost for each color-ID. That is, start from cost 0 and every time the color-ID shows up as one of the edge ends, increase the cost by 1.

2. Start a post-order traversal on the MST. For each node c_i:

   (a) Set the sub-tree cost as the sum of the sub-tree costs of its children added to the local cost of the node: $subtree\_cost_i = local\_cost_i + \sum_{j=1}^{\#children_i} subtree\_cost_{child_j(c_i)}$.

   (b) If $subtree\_cost_i > c$, reset both node costs to 0; add the color-ID to the list of "static IDs".

3. Query the color-IDs in the "static IDs" list and put the color bit-vectors in the "static cache". Since the list has been filled through a post-order walk, the ancestors of each node appear after the node in the list. Therefore, if we construct the color bit-vectors starting from last color-ID to first in the list, we can even use the "static cache" while constructing it.

Interestingly, in all our experiments, we observe a strong correlation between the color-ID query time (negative correlation) and memory (positive correlation) with the value of cost c, the former increasing as the time cost c increases and the later decreasing as fewer nodes need to be cached.

It is important to note that, at query time we still augment this "static cache" with a dynamic LRU cache. For each color-ID, we first look it up in the "static cache", and then, if it is not found, in the LRU cache. If the color-ID is not found

in either, we then traverse to the node's parent and continue the walk until either we reach the root or until we reach a color-ID encoded in any of the caches.

Memory management: Although encoding the color class representation in Classic Mantis using a MST in MST-based Mantis improves the index scalability, both in terms of query memory and disk usage, still the high memory consumption during MST construction remains a bottleneck for scaling the index to more samples. Almost all the steps of constructing the MST are memory-expensive if we prohibit intermediate disk usage. During the construction process, first, we construct a sparse graph of colors by only adding a subset of edges we believe are potentially low-weight edges. There we only add an edge between two colors if their corresponding k-mers are neighbors in the colored de Bruijn graph. This idea is based on the observation that neighboring k-mers in a colored de Bruijn graph tend to have similar colors. Practically, the stated heuristic helps reduce the order of the color-graph edges from $\mathcal{O}(n^2)$ down to $\mathcal{O}(n)$. This change in order, makes the whole MST construction practical in the first place. However, due to the large number of colors, n, the pipeline is still memory-hungry if implemented naively. We explain below a number of optimizations we adopt to improve the construction of the MST:

- *Store serially-accessed structures on disk.* Large structures, such as the color-graph edges, which are accessed once through serialized scanning can be stored on disk. The tradeoff of random access memory for time (and external memory) in such cases seems a practical way to improve scalability.

- *Discard edges with weight greater than a global threshold during edge weight*

calculation for the color-graph. The input list of edges to Kruskal's algorithm [65], which is used to construct the MST, should be sorted based on weights. Since the weight value is bounded above by the number of samples in the index (which we know ahead of time), during the weight calculation process, we perform a bucket sort on-the-fly, moving the edges into the corresponding weight bucket, which is a file on disk. We only store edges with a weight up to a predefined threshold based on the heuristic that, eventually, most of the high-weight edges would be discarded during MST construction. This decision will not affect the functionality of the MST construction, since the presence of the dummy node ensures a connected spanning tree can always be built. Thresholding the input edge weights can only lead to a sub-optimal spanning tree compared to one constructed on the full set of color-graph edges. Practically, we observed only a small effect of this heuristic on the size of the final MST.

- *Store dummy-edges in a different data structure than non-dummies.* As per our definition, one end of a dummy edge is always the zero vector, and as a result, for each dummy edge of $\langle c_i, c_{dummy} \rangle$, the weight is the number of set bits in c_i, and the edge delta is the indices of the set bits in c_i. Hence, we can store these edges, which are a considerable fraction of edges in a data structure specialized for dummies, and keep them always in memory. The structure to store weights is simply a bit-packed int-vector with n slots of width $\lceil \log_2 n \rceil$.

- *Design a memory-efficient structure to store the weighted adjacency list for*

the MST. The result of Kruskal's algorithm is the list of edges in the MST, and each edge's weight, in the form of an adjacency list. The required space to store the adjacency list is linear in the number of nodes (or edges) in the MST (i.e. number of colors in the colored de Bruijn graph). This can be quite costly if implemented naïevely. We adopt an efficient representation of the weighted adjacency list that is explained in supplementary section 5.2.

- *Fill parent vector via a hybrid DFS-BFS walk.* To fill out the parent vector from the adjacency list, we need to perform a DFS or BFS over the tree starting from the root. However, because the MST is large both in width and height when constructed over many samples, both BFS and DFS traversals require substantial memory to perform. A BFS or iterative DFS procedure require a lot of memory for book-keeping, and a recursive DFS would fail as a result of stack overflow. Instead, we use a hybrid traversal to fill out the parent vector. The idea is similar to iterative deepening search [133], keeping the memory constraint in mind. We set a depth limit for DFS. We start a DFS from the root and set the parent-child relationship in parent vector. Every time we reach our the limit, we keep the list of nodes at that limit (similar to the bookkeeping in BFS, but only in one level), and restart a DFS from each node at the level up to the next level where the difference again passes the limit. We continue this, iteratively, until we observe all the nodes in the tree and complete filling the parent vector. Rather than storing the actual IDs of the nodes each time we stop the DFS, we keep a `bv` of size n (the total number of

colors/nodes in the tree) and set the related bits for the nodes that are going to be the root for the next DFS start. In every iteration of the iterative DFS, starting from node i, we reset bit i in the `bv`. We find the set bits in each round by traversing the `bv`. This DFS procedure can also be parallelized.

Parallelization: Given the scale of the data structures being constructed, it is practically important that construction be parallelized even if the construction procedure itself is designed to be efficient. In our implementation, we have kept this in mind, and the following steps of the algorithm have been parallelized:

- *Walking the partitioned CQFs to construct the edges.* If we assume that we have t threads, each partitioned CQF is divided into t equal-size parts. Each thread walks over the k-mers in its designated part of the partitioned CQF searching for the neighbors of the k-mer in the entire partitioned CQF. The edges are stored on disk to keep working memory low. Therefore, we make use of a multi-producer queue to store batches of edges on disk every often.

- *Calculating edge weights.* We simply divide the edges between threads. Each thread is responsible for calculating the weights of the assigned subset of edges. For this, each thread needs to access the MST structure, and the associated structures mainly the static cache. The static cache is constructed only once before the start of this step and will not be modified at any subsequent point, so it can be safely shared across threads to read from. When storing these edge weights (temporarily) to disk, we again need to buffer the edges and then flush them to disk in batches. Practically, we have observed considerable

unevenness in the distribution of edge weights. The vast majority of edges have small weight values and the weight distribution is highly-skewed. For example, for an index over $2k$ samples with close to 1 billion edges, less than $200,000$ of them had a weight > 1000. To keep the balance in frequency of flushing the buffers for each weight bucket, we used a geometric distribution (common ratio $= \frac{1}{2}$) of the buffer sizes where the files assigned to smaller weights have larger buffers that take up more of the allocated space in RAM.

- *Filling Delta and Boundary vectors.* Although the process of querying the input MSTs is the same as calculating the weights, there is an additional complexity filling the delta vector. The size of the delta vector and the order of the deltas are already predetermined; the size is total weight of the MST and the order is the same as the edges in parent vector. This means that each thread can be assigned a start and end of the range it is allowed to fill in the delta vector. The goal is achieved by performing one extra pass over the parent vector to sum up the weights of all the edges belonging to the same thread segment, assuming each thread is responsible for extracting deltas of edges in a consequent section of the parent vector.

2.3 Constructing and merging minimizer-partitioned counting quotient filters

The efficiency of the MST-based color representation shifts the memory bottleneck from the color classes to the counting quotient filter representation itself, which now

represents the dominant part of the data structure. One immediate solution for the case of serial access to the counting quotient filter is to use `mmap` to let the system page in the required memory and simultaneously "suggesting" it free the previous pages from the counting quotient filter while iterating over the structure. Specifically, such a design allows to prevent loading 3 large counting quotient filters at the same time in memory during the counting quotient filter merging process as that merge operation traverses the input (and output) counting quotient filters in order. However, such a solution is inadequate when random-access to the counting quotient filter is needed, as is the case during MST construction and later during index query.

To circumvent the need to hold a single counting quotient filter containing all k-mers in memory at once, we make use of minimizers [135] to separate k-mers into smaller blocks of counting quotient filters. Defining an $\updownarrow - mer$ as a sub-string of length l in a sequence of length k where $l < k$ and considering some ordering for the $\updownarrow - mers$, A `minimizer` of the sequence is formulated as the smallest $\updownarrow - mer$ of all the sequence $\updownarrow - mers$ [135]. This concept is commonly used for partitioning sequences (specifically k-mers) into smaller groups with smaller memory consumption for different tasks such as assembly [31], and alignment [70, 84, 117, 161]. For this type of sequence, the minimizer is defined over the canonical form of the k-mer (smallest of the k-mer sequence and its reverse complement). The main benefit for this type of partitioning is that the partition ID for a k-mer can be retrieved only by having access to the k-mer sequence. We basically adopt such an approach for partitioning the counting quotient filter into smaller components which

we call partitioned CQFs, where all k-mers with the same minimizer go to the same partitioned CQF. To query a k-mer in the new data structure, we first calculate k-mer's minimizer value and query the partitioned CQF that represents the minimizer which results in reducing the query memory from the size of a counting quotient filter down to the largest partitioned CQF's size. To keep the distribution of k-mers into minimizer blocks close to uniform we use a random order for the minimizers.

One could, in theory, create a separate partitioned CQF corresponding to each minimizer, but this approach has two main issues: (1) it requires storing $4^{\updownarrow}$ partitioned CQF files on disk for partitioning based on a minimizer of length l; and (2) it may lead to partitioned CQFs with very small numbers of k-mers, which leads to small counting quotient filters with few slots and large remainders. This would in turn, result in using the counting quotient filter in its most inefficient way. To overcome these issues, we place the k-mers for multiple consecutive minimizers into the same partitioned CQF. Assuming we know the distribution of minimizers across k-mers ahead of time, a partitioned CQF, $pcqf_p$, is defined as follows: For a threshold value t (the maximum number of k-mers allowed in each partitioned CQF), $pcqf_p$ contains all k-mers with a minimizer in the range of $[m_i..m_j)$, where

- $\forall\, k_q \in \text{pcqf}_p\ ,\ \forall\, k_t \in \text{pcqf}_{(p+1)} \quad (\text{minimizer}(k_q) < \text{minimizer}(k_t))$,
- $m_j > m_i$ and
- either $m_j = m_i + 1$ or $|\text{pcqf}_p| < t$, where $|\text{pcqf}_p|$ is the total number of k-mers in pcqf_p

The above conditions guarantee that no partitioned CQF contains more than

t k-mers, except for the special cases where a single minimizer represents a larger number of k-mers than the threshold. Each partitioned CQF may contain a represent range of minimizers based on the distribution of k-mers across minimizers. We store the range as auxiliary information which is part of the index. These modifications solve the two main issues that would arise by naïvely assigning each minimizer to its own partitioned CQF, but this construction can still not be effectively used for MST construction since it does not support efficient query for the neighbors of a k-mer.

For building the color-graph during the MST construction, we need to access all the neighbors of a k-mer which might not have the same minimizer value of the k-mer and thus be in a different partition than the k-mer. In such cases, we need to search for the neighbor in its corresponding partition which requires loading a potentially different partition from disk. We tackle this problem by adding one more condition to the definition of a partitioned CQF. For any k-mer whose minimizer is assigned to pcqf_i, all the neighbors of the k-mer should also be in the same partitioned CQF. If the neighbor k-mer's minimizer is one of the minimizers the covered by $pcqf_i$, then it already ends up in the same partitioned CQF, otherwise, we would insert the neighbor k-mer into $pcqf_i$ in addition to the partitioned CQF that it belongs to based on its minimizer value. This requires that some of the k-mers are duplicated and put in two different partitioned CQFs, one that they belong to based on the minimizer value and one that their neighbor k-mer belongs to. To determine the duplicated k-mers we follow the same process explained by Chikhi et al. [31] for compacting the non-branching paths in a de Bruijn graph in a memory-

efficient manner, which requires the same constraint to be satisfied. With this final modification, the color-graph construction that is required for building the MST is made efficient.

2.3.1 Merging partitioned CQFs

The process of merging two sets of partitioned CQFs is essentially the same as that of merging two counting quotient filters, which is described in section 2.1. We follow the same steps, walking the two input lists of partitioned CQFs in order:

1. Sample for popular color pairs for the union of the k-mers in left and right input and put them in a map with key of a color-pair to value being the abundance of the pair in the sampled list.

2. Store the rest of the color pairs to disk.

3. Construct a MPH over the list of unique color pairs.

4. Finally, merge the two inputs' partitioned CQF lists into the output partitioned CQF list which contains the union of the k-mers in the left and right and the associated color ID. For each k-mer's new color ID, follow the same protocol as in Classic Mantis merge (i.e. look up the color pair in the popular color map first and if not found, construct the ID via the MPH).

After merger, the output partitioned CQFs can be used for MST merger as explained in section 2.2. The one main difference between partitioned CQF and regular counting quotient filter merging is how to find the union k-mers in a list of

partitioned CQFs compared to a pair of regular counting quotient filters. In Classic Mantis this process was as straightforward as walking two sorted list and outputting the union of them into another list which is by definition also sorted. In the new scheme, if we had the one-to-one relationship between a partitioned CQF and a minimizer, the process would be the same as walking one counting quotient filter, except now we would be walking over multiple counting quotient filters in order of minimizers. Each partitioned CQF with the same minimizer from the two inputs would be loaded into memory, and we would store the union of the k-mers from both into the output partitioned CQF corresponding to the same minimizer. When multiple minimizers are assigned to a single partitioned CQF, this process is not trivial.

Based on the first item in the definition of a partitioned CQF in 2.3, the partitioned CQFs are sorted based on minimizers so that for $i < j$, all the minimizers that map to $pcqf_i$ are essentially smaller than the minimizers in $pcqf_j$. However, there is no guarantee for any specific relationship between the two input Mantis partitioned CQFs with the same ID. Thus the intersection of the minimizers each of the two partitioned CQFs cover could be anything from zero minimizers to all minimizers of one of the two input partitioned CQFs. This makes it difficult to walk any pair of partitioned CQFs to obtain the union list. At the same time, we know that each $pcqf$ is a cqf containing k-mers sorted based on their hash values, so if a partitioned CQF contains a range of minimizers, the k-mers with different minimizers are interspersed within the partitioned CQF, without any guarantee that the k-mers of the same minimizer appear next to each other. Figure 22 shows

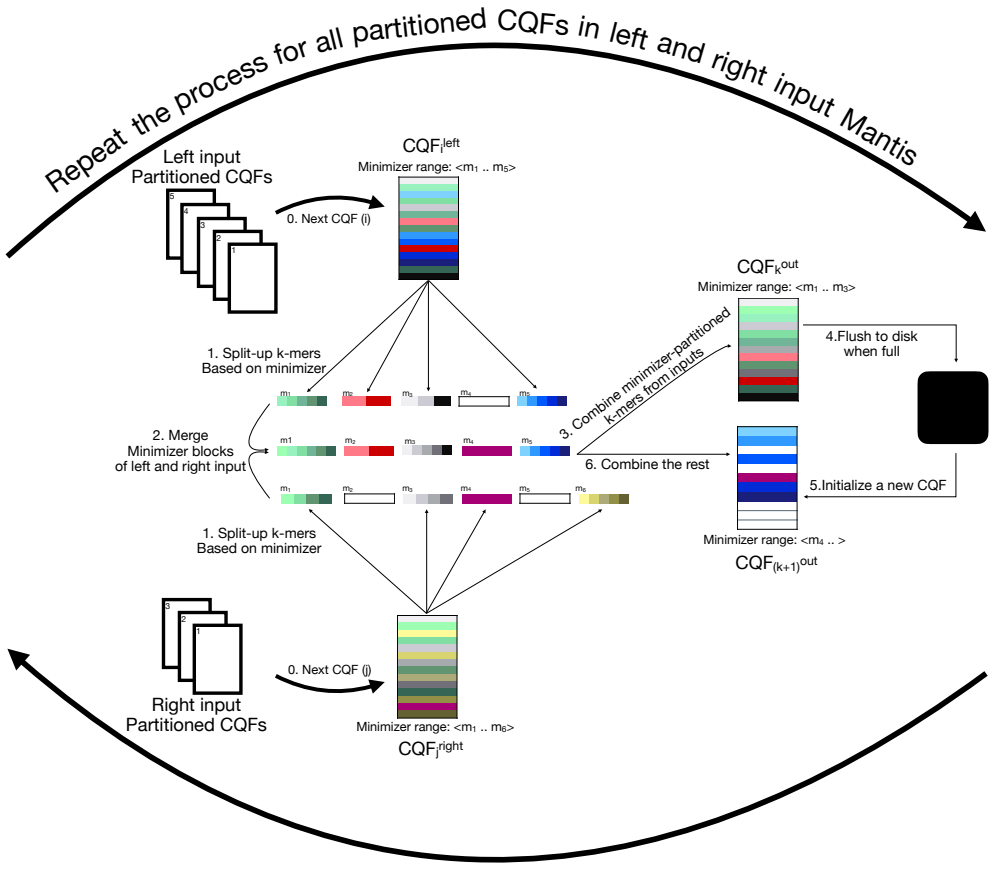


Figure 22: A toy example, illustrating the steps for merging two partitioned CQFs. The two arrows on top and bottom are indicative of the loop over each input partitioned CQF. For each partitioned CQF in each of the left and right inputs, first the k-mers are parted based on their minimizers. For each minimizer that its associated k-mers have been processed in both inputs, we merge the k-mers of the two input minimizer buckets. Then we walk over all the minimizer buckets for which we have the merged result, and insert the $\langle \text{k}-mer, color \rangle$ pair into output counting quotient filter. Anytime the counting quotient filter is full, we flush it into disk, reinitialize a new partitioned CQF and continue inserting into the new partitioned CQF. We may reach the end of our merged minimizer buckets in the current round while our output counting quotient filter is not yet full; in that case, we continue filling it in next iterations.

the steps required to merge k-mers from two input partitioned CQFs into one (or possibly more) output partitioned CQF (s). In addition, we provide the pseudo-code for the merge operation in algorithm 2 in the supplementary material.

The methodology is as follows. To begin, we load the first partitioned CQF from left and right input into memory and separate the k-mers in each into their corresponding minimizer block by a linear traversal of the partitioned CQF, finding

the minimizer of the k-mer and inserting it into the minimizer block (step 1 in figure and line 61 of the algorithm, procedure "walk_partition"). Since the k-mers are sorted by hash values, it is guaranteed that the k-mers in each minimizer block are also sorted. At this point, we are in the case of one-to-one map from a minimizer to the list of k-mers. Now, on a second linear pass over the k-mers per each minimizer in the intersection of the completed minimizer blocks in the left and right input, we find the union of the k-mers and put them in the union block with the same minimizer value along with their color IDs (step 2 of the figure and line 8 of the algorithm, procedure "compare_sorted_list"). At this point, we can follow the same steps as in Classic Mantis, either store the color pairs for the union k-mer list or store the k-mer and color ID into the output counting quotient filter (procedures "find_uniq_colorPairs" and "store2cdbg" in the algorithm). We fill the output partitioned CQF walking the merged minimizer blocks in order throughout steps 3 to 6 in the figure. During the traversal, we flush each output partitioned CQF to disk when we can no longer add a new minimizer since it would pass the threshold. We also store the associated minimizer range to the output partitioned CQF and reinitialize a new empty partitioned CQF for the rest of the minimizer blocks. We free the memory from each minimizer block as soon as we are done processing it.

Memory requirements: Since the connection point of the left and right inputs are the minimizer blocks rather than the partitioned CQFs themselves, we do not need to load two partitioned CQFs into memory at the same time. We switch between inputs looking at the maximum minimizer covered by each input. If we have more

minimizer blocks from left input, we load the next partitioned CQF from the right input and vice versa. This strategy guarantees that, at each point of the process, the total number of k-mers loaded into blocks does not pass twice the threshold for a partitioned CQF because at each point, we make sure that we can get rid of a subset of minimizer blocks by carefully choosing which input to process. Also, we would have at most two partitioned CQFs in memory when filling the output partitioned CQF (one input counting quotient filter and the output counting quotient filter). Altogether, the total memory requirement for the procedure based on the threshold t for a partitioned CQF is $2 * t * sizeOf(kmer, colorID) + 2 * sizeOf(pcqf)$.

Parallelization : A single partitioned CQF is simply a counting quotient filter, and we can divide the range of hash values in a counting quotient filter into t equal size segments (t being number of threads given by the user) and let each thread separately walk the assigned segment and collect the k-mer and color pairs and partition them into their associated minimizer block. Walking the partitioned CQF and collecting the k-mers into different minimizers is performed many times throughout the merging process, and making this process work in parallel has a large effect in the final performance of the merge.

3 Evaluation

3.1 Experimental Setup

3.1.1 System Specifications

As with prior work [8, 124], the input to Mantis is a list of *squeakr* files [121]. The squeakrs are each constructed from a specific `FASTQ` file selected from human RNA short read sequences publicly available via the SRA [81]. The system for all the experiments is an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD running Ubuntu 16.10 (Linux kernel 4.8.0-59-generic). Constructing and merging Mantis were both conducted using 16 threads. However, VARI only support a single-thread and therefore, was benchmarked on a single thread. Query benchmarks were performed using a single thread for both tools. Specific to VARI, since the limit on the number of samples is hard-coded as a compile-time constant in the Makefile (which has a direct effect on the data structures allocated and thus the construction/merge memory consumption), we have used a distinct executable file for each of our experiments. This was done to minimize the memory and disk use of VARI for each of the experiments (rather than using one executable with this constant hard coded to the largest number of samples).

3.1.2 Input Data

For the input to our experiments, downloaded 15,000 FASTQ files from NCBI [116]. The list of accessions can be accessed through github repository of the project from file "shuffled_10k_paired.for-recomb". In the first step, we needed to construct the Squeakr files for all the samples. This step was performed on a cluster of 150 machines roughly three weeks. For each file, only the k-mers with abundance value more than a predefined threshold are selected. The value of the threshold is decided based on the size of the FASTQ file (gzipped). This prefiltering step is useful to eliminate spurious k-mers that occur with insignificant abundance and has been adopted from the original SBT paper [148]. The k chosen for the k-mers across all the experiments is fixed at $k = 23$. The total space required to store all FASTQ files and their corresponding Squeakr file is $2.9TB$ and $970GB$ respectively.

3.2 Merging Benchmarks

For each of the merging experiments of two Mantis with n and m samples, we construct the n and m samples first and then execute the merging procedure over the two. To save the runtime for all the experiments, at each level of merging except for the first, the left input is the results of the previous merge. For example for the results of the merge into $5k$, we merged the mantis over $2k$ samples from the previous merge with the mantis on the remaining $3k$ samples. We benchmark the max RSS value for merging memory. Unfortunately we were not able to perform Vari merge further than $2k$ samples. Constructing Vari index requires a massive amount of

disk space because they utilize the external sorting algorithm implemented in stxxl library which starts from a user-defined size on disk and extends the size during the process if required (this could be due to the specific settings defined in Vari). The final stxxl file size after constructing the Vari index over $1k$ samples was over $2TB$. We dedicated a disk of $3.5TB$ to these experiments which was not enough for the experiments above $1k$. One thing we noticed is the lower intermediate disk usage of Varimerge compared to Vari itself. Therefore contructing many Varis over small subset of samples and then building the final Vari over all via recursive merge could be a solution to get the results. In fact, we use the same pipeline for building the Mantis with partitioned CQFs. Constructing both Vari with and without merge, the size of the indices are slightly different for smaller experiments. Running Varimerge recursively to construct larger samples, we might observe more divergence between the size of the final indices via merge vs constructed from scratch. However, this and the longer construction time are the two things that must be tradeoff for the gigantic disk required to enable scaling Vari index to larger samples. As the results show, Mantis merge does a better job with respect to all three metrics and the results become further running merge over larger number of samples. Looking at the memory results, it seems unlikely that Vari would be able scale to $10k$ samples due to the super-linear growth in required memory.

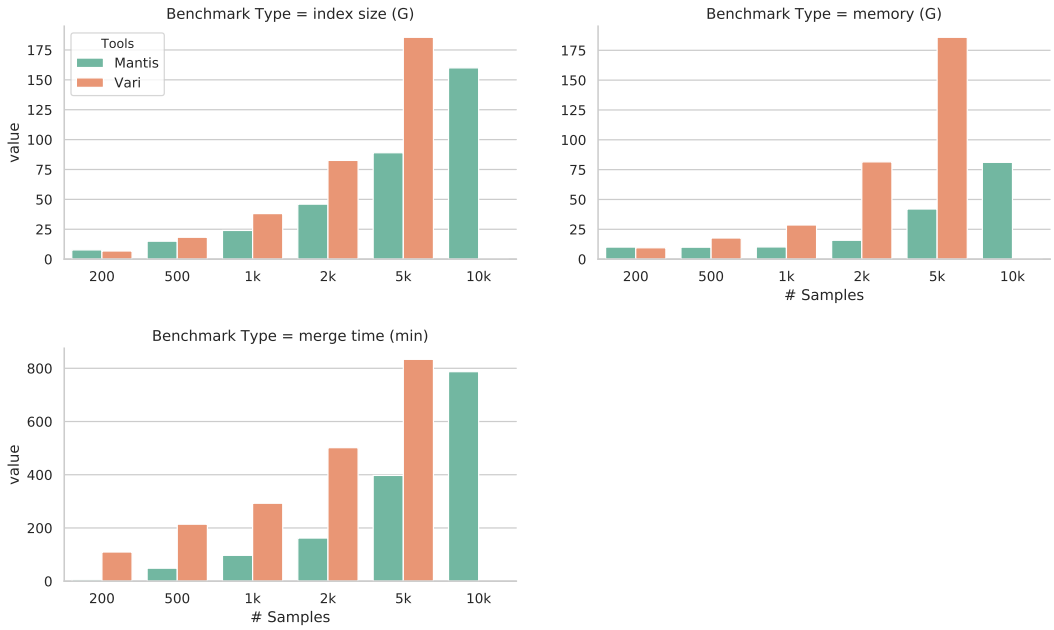


Figure 23: Benchmarking Vari and Mantis merge for building different number of samples from 200 up to 10k. We were unable to construct input Vari indices for merging into 10k samples. The results show Mantis merge is superior in all the metrics; specifically the construction memory. Mantis's required memory for merging two 5k samples into a 10k one is similar to merging two 1ks for VariMerge. Moreover, the size of the VariMerge index for 5k is already larger than the space required for Mantis index on 10 samples.

3.3 Query Benchmarks

For evaluating the query performance, we compared the query time and memory requirement over the current Mantis index with partitioned CQFs with the MST-based Mantis, which has already been shown to be among the sequence search indices with the fastest search capabilities. We have performed query for both the index described in this paper and MST-based Mantis in bulk mode, and have benchmarked the total query time including the index loading time (but using a warmed-cache — each experiment was run twice, with only the second time being recorded).

In bulk query mode, we extract the distinct k-mers of all the sequenced queries,

and find the color-ID and the corresponding color bit-vector for each which signifies the experiments in which they are present. This allows us to have a map from each distinct k-mer to the list of queries that contain it, which we can use to find the total number of k-mers in each query that are present in an experiment. In a bulk query procedure, each partitioned CQF is loaded only once and each color bit-vector is also constructed only once. As the plots show, the query time is very similar to that of MST-based Mantis while the memory is more efficient (up to $7x$). The partitioned CQF design, however, is not as efficient for continuous single queries as it is for bulk except if all the partitioned CQFs are loaded into memory (which eliminates the memory benefit). In fact, assuming a cold-cache run of the query process, on the system we ran the experiments on with an I/O bound $\sim 800Mbps$ it takes ~ 12 seconds to load each partitioned CQF which is of size $1.2G$. Therefore, for example for an index on $10k$ samples, it would take more than 30 minutes to just go over the partitioned CQFs looking for the k-mers. Repeating the process for each single query does not seem efficient. One interesting idea to speed up individual query time is to use a distributed systems, loading some subset of partitioned CQFs on each, and broadcasting the query. This idea can both maintain a low-memory requirement for each individual system as well as allowing fast bulk and individual queries.

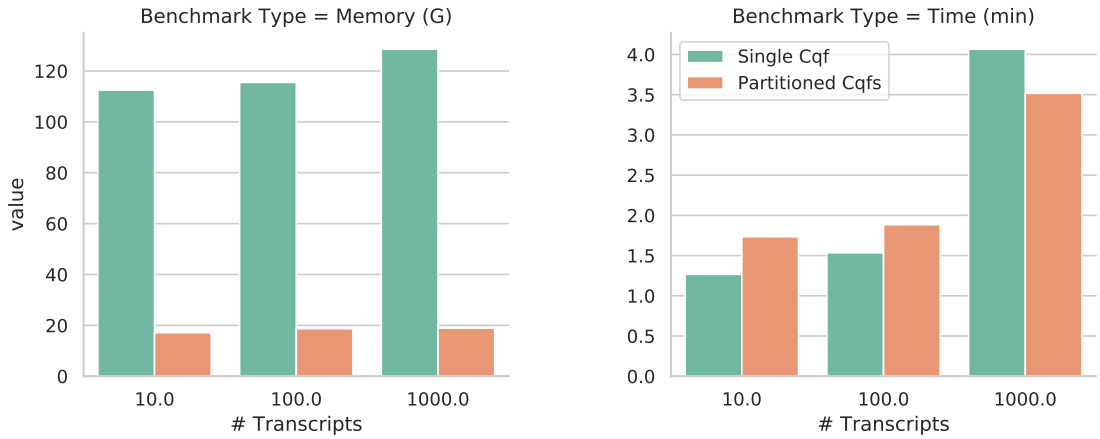


Figure 24: Comparing query performance of the new Mantis with partitioned CQFs and the Mantis with one giant CQF in a warmed-cache state. The query time is similar while the query memory is now limited to the MST size rather than the giant CQF.

3.4 LSM-Tree Benchmarks

The Mantis-based LSM tree consists of a sequence of levels where each level i either contains a Mantis index M_i, or is empty. The depth of the LSM tree is d if M_{d-1} is the highest-level nonempty Mantis index in the tree. Each of these indices in the tree covers a disjoint subset of the total samples set. We also maintain a "RAM" level L_r in the LSM tree that does not contain a Mantis index, but rather stores the CQF files as is, of the samples corresponding to this level. Conceptually, this "RAM" level sits atop the entire tree, and is usually bounded to store a small number s of samples. Cumulatively, the "RAM" level r and the Mantis indices $M_0, \ldots, M_{d-1}$ cover the entire set of samples.

If the LSM tree contains d levels with the mantis indices $M_0, \ldots, M_{d-1}$ (except M_{d-1}, some of the rest might be empty), each non-empty level contains an index on an increasingly larger subset of the total sample set. Each level has a maximum allowed size, defined by a threshold parameter t and a scaling factor (fanout) pa-

rameter c. Specifically, we bound the size of Mantis indices at each level i using the number of partitioned CQFs that this level contains, defining the maximum CQFs count to $(c^i \times t)$. The t parameter is the maximum number of CQF files to keep at the lowest level index M_0, and c is the growth factor of the tree. When a new sample is to be added to the LSM tree, it is put into the "RAM" level L_r, and if the number of samples at L_r exceeds s, then a Mantis index M_r is built from these samples, L_r is emptied, and M_r is merged into M_0. A propagation of a Mantis index from a level i to level $(i+1)$ is triggered when the number of CQFs at the index M_i exceeds $(c^i \times t)$, and the index M_i is merged into index M_{i+1} of level $(i+1)$.

For our benchmarking purposes, we used the "RAM"-level threshold as $s = 100$ samples, the level-0 threshold as $t = 5$, and the scaling factor as $c = 4$. To evaluate the scalability of Mantis merge beyond $10k$, we started the LSM-tree updating experiment from the available Mantis index over $10k$ at previous step. We calculated the level that the index belongs to considering the size of the index and assumed the LSM-tree be occupied at that level by a Mantis with partitioned CQFs over $10k$ samples. Figure 25 shows the memory consumption and time for adding the sample after every 100^{th} insertions. That is the insertion that triggers the compaction procedure. As the results show we observe a steady behavior in insertion time for the first 4000 samples after $10k$. We observe a close to constant insertion for most of the samples and there is a regular peak of longer time ($\sim 20,000$ seconds) about every 500 samples. That is when we need to continue the merge more than 1 level up to level 2. There is a big peak $\sim 4000^{th}$ insertion when the cascading merge will go one level deeper into level 3 where everything gets merged into the $10k$ index.

These spikes also show up in the cumulative time plot as the steps that impact the close to linear total time increase over number of samples (which would again mean a constant time insert for other samples except for those that cause the steps in the cumulative plot). The memory story is however different as it is more steady and close to constant for all the insertions except for the one that requires the big merging. At that point the memory increases based on the size of both number of samples i hand as well as the $10k$ index sitting on disk. The bottleneck in this process happens during the merging process and still has space to improve. Overall, we can say that the results are promising and as follow the expectations based on the definition of the LSM-tree.

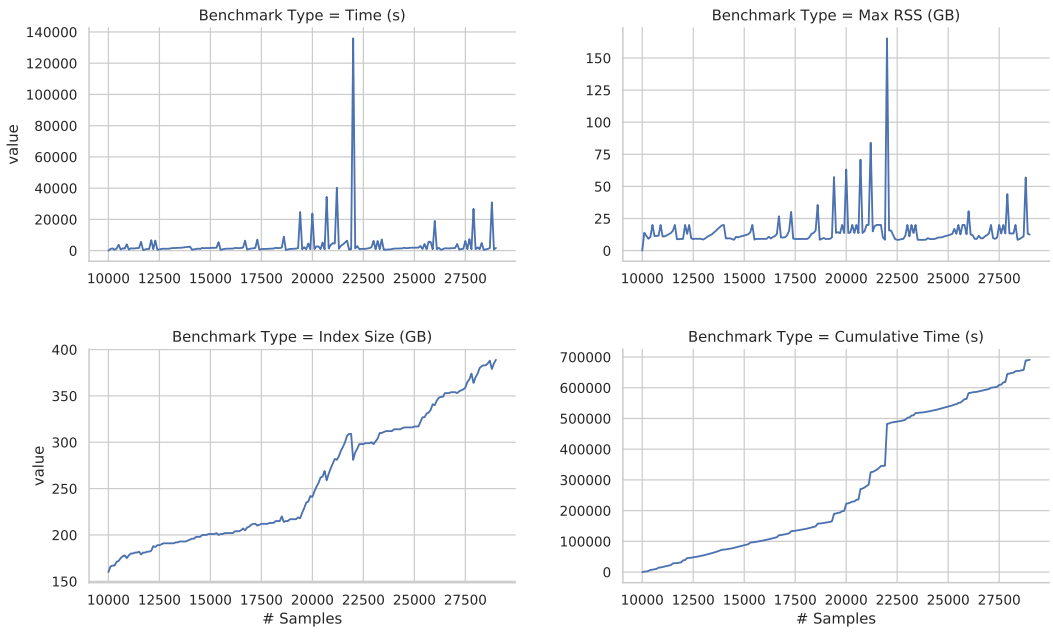


Figure 25: Performance of the LSM-tree update process starting from an LSM-Tree with 10k samples and adding batches of 100 samples up to 29k. The spikes in time and memory show up when the cascading merge happens with deeper and thus larger index merging. The cumulative construction time otherwise is linear.

4 Discussion and Conclusion

In this work, we have described an incrementally-updatable sequence search data structure that maintains many of the favorable properties of the previous index (MST-based Mantis), but also advances the index with respect to its scalability and memory requirements. By incorporating the Mantis index into an LSM tree structure, we enable insertion of a new experiment without requiring reconstruction of the index. We initially add experiments with the cost of only constructing a Squeakr in memory. In cases where the number of Squeakrpasses a threshold, a cascading process of index creation and merging takes place. We provide a memory-efficient and highly-parallelized merging algorithm for the direct merge of the two (partitioned CQF-based) MST-based Mantis inputs. This greatly reduces the required disk space compared to merging Classic Mantis inputs, which requires the creation of a large color class matrix. This reduced intermediate disk usage is important for scalability itself, since even if we were to use the MST-based Mantis at query time, without direct merge of the MSTs, index construction would require the creation of very large intermediate color class tables, which would dominate the storage requirements for the index.

We also replace the counting quotient filter filter in Mantis with a collection of partitioned CQFs that are (individually) much smaller in size and can be loaded individually during the MST construction and query processes. This substantially reduces the query memory requirements significantly and ensures that the counting quotient filter data structure is no longer the memory bottleneck in MST construc-

tion and merging process anymore. As observed in the benchmarks, for bulk query, the memory requirements are reduced by $7x$ for an index on $10k$ samples without increasing the overall query time. We compare the merging of our new data structure witht he merging procedure introduced in Varimerge, that to our knowledge, is the only colored de Bruijn graph representation which can be updated via merge (Varimerge). We find that the construction time, memory, and final index size of our new index are smaller than those of Varimerge. We also provide benchmarks for incremental updating of our data structure using LSM trees to show the amortized cost of adding new experiments to our data strucutre. Using this procedure, we were able to construct an LSM tree index over $15k$ raw sequencing samples which takes $375G$ of space and can be extended to larger number of samples.

As future work, we would like to explore building a distributed index that would allow indexing all unrestricted human RNA-seq data in the SRA which could then be used to produce a system for online query. One property of the partitioned CQF in the new Mantis representation is the ability to easily distribute the index across multiple systems with relatively low memory and cpu requirements. As the partitioned CQFs are distinct, for each query sequence, the k-mers can be distributed to the corresponding sub-indices. Further, the merging can happen under the same circumstance where the k-mers of the two inputs can be separated based on minimizers and merged in different systems in parallel. The merging process however, would require considerable network bandwidth, which raises the question of which Mantis data structure one is best suited for a distributed system (e.g. one could instead maintain a completely separate MST-based Mantis indices on each

distributed node). We note that the query in our LSM tree structure depends on the depth of the tree and also the hardware used to support the out-of-RAM level. In particular, the use of SSD hard disks would improve the performance considerably by reducing cost of I/O access to the indices in higher levels of the tree that are stored on disk. Comprehensive benchmarking of the query time over different types of hardware is left as future work. Finally, one may also think of some smaller-size approximate data structure that would point the k-mer directly to the level that it should be search in.

The incrementally-updating Mantis is written in C++17 and is available at `https://github.com/splatlab/mantis/tree/mergeMSTs`.

5 Supplementary Material

5.1 partitioned CQF merge pipeline

5.2 Detailed design of the memory-efficient structure to store the weighted adjacency list for the MST

As explained in section 2.2, the MST merge process is highly memory-intensive. We pointed out some of the optimizations we consider during the implementation to improve memory consumption, one of which was using a succinct representation of the weighted adjacency list. Here we explain this idea in more details.

By the end of the MST finding process, we would have a list of selected edges and their weights. As shown in figure 21, the next step is assigning directions from

Algorithm 2 Merging two sets of partitioned CQFs from left and right colored de Bruijn graphs ($cdbg^l$ and $cdbg^r$) into output colored de Bruijn graph ($cdbg^o$). Code simplified.

```
colorMap : MAP(⟨color^out; ⟨color^left, color^right⟩⟩)
procedure MERGE_PCQFS(cdbg^l, cdbg^r, cdbg^o)
    COMPARE_SORTED_LIST(cdbg^l, cdbg^r, find_uniq_colorPairs)        ▷ Fills colorMap
    COMPARE_SORTED_LIST(cdbg^l, cdbg^r, store2cdbg)                  ▷ Uses colorMap
end procedure

kcList^i_m: List of ⟨kmer, color⟩ pairs in a CQF partition for input cdbg^i where mnmzr(kmer) = m.
procedure COMPARE_SORTED_LIST(cdbg^l, cdbg^r, process)
    minMinimizer ← 0
    for p in MAX(# of counting quotient filter partitions for cdbg^l and cdbg^r) do
        for i in l, r do maxMinimizer^i ← cdbg^i.WALK_PARTITION(p, kcList^i)
        end for
        maxMinimizer ← MIN(maxMinimizer^l, maxMinimizer^r)
        for m in minMinimizer..maxMinimizer do
            kc^l = kcList^l_m.GETCURRENT()                           ▷ kc = ⟨kmer, color⟩
            kc^r = kcList^r_l.GETCURRENT()
            repeat
                if kc^l.kmer < kc^r.kmer then
                    process(kc^l, NA)                                ▷ NA: Not Available
                    kc^l.NEXT()
                else if kc^l.kmer > kc^r.kmer then
                    process(NA, kc^r)
                    kc^r.NEXT()
                else
                    process(kc^l, kc^r)
                    kc^l.NEXT()
                    kc^r.NEXT()
                end if
            until kc^l.HASNEXT() or kc^r.HASNEXT()
            DELETE(kcList^l_m)
            DELETE(kcList^r_m)
        end for
        minMinimizer ← maxMinimizer
    end for
end procedure
```

```
procedure FIND_UNIQ_COLORPAIRS(kc^l, kc^r)                          ▷ kc = ⟨kmer, color⟩
    if ⟨kc^l.color, kc^r.color⟩ ∉colorMap then
        index ←colorMap.length
        colorMap.ADD(⟨kc^l.color, kc^r.color⟩ → index )
    end if
end procedure

procedure STORE2CDBG(kc^l, kc^r)                                    ▷ kc = ⟨kmer, color⟩
    colorID ←colorMap [⟨kc^l.color, kc^r.color⟩]
    if kc^l! = NA then
        kmer ← kc^l.kmer
    else
        kmer ← kc^r.kmer
    end if
    cqf^o ← cdbg^o.CURRENTPARTITION()
    if cqf^o.ISFULL then
        cqf^o.STORE2DISK()
        cqf^o ← cdbg^o.NEWPARTITION()
    end if
    cqf^o.INSERT(⟨kmer, color⟩)
end procedure

l : minimizer length, "8" in our case
procedure WALK_PARTITION(p, kcList)                                 ▷ p: partition ID
    pcqf_p ← LOAD p^th CQF partition from disk
    if pcqf_p == NULL then
        return MAXINT
    end if
    for kc in pcqf_p do
        m ← FIND_MINIMIZER(kc.kmer)                                 ▷ (Between 0 and 4^l)
        kcList_m.INSERT(kc)
        maxMinimizer ←MAX(maxMinimizer, m)
    end for
    return maxMinimizer
end procedure
```

parents to children in the tree starting from the dummy-node as the root for which we need a constant access from any node to all its adjacent ones. As we are storing the adjacency for a tree which is the most sparse possible graph representation for n nodes, we would choose the adjacency list over matrix. Although the order for storing the tree adjacency list is $\mathcal{O}(n)$, the n is considerably big and with high growth rate over samples that implementation-wise, the constant for the n also matters. In a naive implementation, storing a weighted adjacency list of a tree with n vertices assuming k bytes to store an empty list in the language (at least 16 to store the start pointer and size) and the largest word size (8 bytes) to cover color-IDs greater than 2^{32} as well as a word size of 4 bytes for weights to cover number of samples greater than 2^{16} (both of which emerge in the scales we run Mantis on), requires $2 * n * (16 + 8 + 4)$ bytes. For example for indexing $80,000$ samples with $n \sim$ 4e9, the required memory would be $\sim 208GB$. We note that this adjacency list should be in memory during the time filling the final structure of the MST which itself takes space.

We design a more thoughtful succinct representation of the adjacency list which eventually reduces the constant noticeably so that in practice the memory for that section is reduced in orders of magnitudes while still allowing constant-time access from each node to its neighbors. Assuming to have a tree with n nodes where node IDs are in the range of $(0..n-1)$, we define the adjacency list through four succinct vectors; two of which are used to store information for the smaller end of an edge $\langle c_i, c_j \rangle$ and the other two for the larger end. We mention that since we do not have self-loops in the tree (edges with equal end IDs), therefore, for n edges

each of the four vectors are of size n. However, the width of the elements in each vector is different and basically the main reason that results in a total allocated space reduction.

In the first vector, nei_{sm_end}, we store the IDs of the adjacent nodes of each node, if the node ID is smaller than its adjacent node ID, in sorted order of the node IDs along with the weight of the edge in a succinct form. Each word of the vector is of width $\log_2 n + \log_2 max(weights)$ bits where $max(weights) = s$, total number of samples. Since nodes can have different degrees as well as different number of connected nodes with the described condition, it is required to store the index of the start of the neighbor list for each node in nei_{sm_end}. That takes us to the second vector, $start\_index_{sm\_end}$ in which at index i, we store the starting index of neighbor list for node i in nei_{sm_end}. In this way, to look up the neighbors of node i and fetch the weights, we first fetch the start index of node i at index i of the $start\_index_{sm\_end}$, say its value is $start_i$; then jump to index $start_i$ in nei_{sm_end} for the first neighbor of node i. The count of the neighbors with greater ID value for each node i is calculated by subtracting the start index of neighbors for i and $i + 1$. If $start\_index_{sm\_end}[i] == start\_index_{sm\_end}[i + 1]$ this means $\nexists edge = \langle c_i, c_j \rangle \mid c_i < c_j$.

The other two vectors, nei_{gr_end}, $start\_index_{gr\_end}$ follow the exact same results for storing the neighbors of node i with IDs smaller than the ID of the node. In this way, we still are storing adjacency list for all the nodes in the tree to support constant-time access. However, we store the edge weights only in one of the vectors, for example in our case only in vector nei_{sm_end}. The memory consumption in

this succinct representation would be $n * (\log_2 n * 2 + \log_2 max(weights))$ for the first pair vectors plus $n * (\log_2 n * 2)$ for the second pair of vectors with total of $n * (4 * \log_2 n + \log_2 s)$ which in the same example of indexing $80,000$ samples with $n \sim$ 4e9 would result in $\sim 67GB$. We can still improve this design by replacing the vectors indicating the start indices of neighbors for each node with a `bv` of size n and a rank data structure on top of it which in the same example would reduce the total memory down to $\sim 38GB$ which is 0.2^{th} of the naive implementation memory.

Chapter 7: Conclusion

The main focus of this work and of my PhD has been on the design and development of reference-based and reference-free sequence indexes. When working at the scale of whole genomes, populations of genomes, and collections of raw seqeuncing samples, the problem of extending indexing strategies to graphs becomes very important. In this document, we presented three data structures for indexing a collection of genomes, transcriptomes, or sample reads in the form of a colored de Bruijn graph or a compacted colored de Bruijn graph. In developing all of the indices presented in this work, the main focus was to achieve scalability over more and larger input sequences or samples while maintaining a reasonable construction and query memory and providing and high-speed queries.

Pufferfish is our index designed for a collection of reference sequences. We extend the index into a full-fledged aligner, Puffaligner, by adding the required steps to find the best chain of uni-MEMs found in the index and by aligning the gaps between the exact matches on the chains. We have also developed Rainbowfish and Mantis as two different indices over a collection of short read sequences. They both demonstrate outstanding performance compared to prior work. We described numerous improvements upon the original Mantis index, which make it both more

space-efficient and scalable in the work on MST-based Mantis, and partitioned CQF-based Mantis. We also add the support for low-cost insertion of new samples by integrating Mantis into the dynamically updatable LSM-tree framework.

1 Reference-based Indexing

In chapter 2 I describe our reference indexing approach, Pufferfish. The index makes use of minimum perfect hashing, and uses succinct representations where applicable to reduce the final size of the index. The index shows lookup performance comparable to traditional hashing-based implementations while using considerably less space. We propose two variants of the Pufferfish index; dense and sparse. In the sparse variant, the user can trade off query speed for index size (i.e. query memory). This variant is enabled by a sampling scheme that take advantage of the unique successor / predecessor relationship between k-mers in a unitig, and allows for fast search on large reference sequences. The desirable properties of the Pufferfish index make it an ideal foundational data structure on which to build a seqeuence aligner. PuffAligner begins read alignment by collecting unique maximal exact matches (uni-MEMs) which are extracted by querying k-mers from the read in the Pufferfish index and extending them to maximality. Then via a dynamic programming approach adopted from Minimap2 [], PuffAligner finds the best chains of uni-MEMs and later aligns the gaps between the chains of exact matches. Through out the process, PuffAligner uses various heuristics for reporting a likely set of alignments per read. The results demonstrate PuffAligner as a highly accurate and

fast alignment tool modest memory requirements. PuffAligner is particularly useful for indexing and aligning to a highly similar collection of sequences, potentially making it a powerful approach in metagenomic analyses, which we also demonstrate via a set of experiments.

A Multi-Purpose Index and Aligner: The main advantage of a data structure like Pufferfish compared to a linear index is the ability to efficiently map reads to a population of genomes or individual genomes with annotated variants. Current tools that are used for alignment and mapping are either mostly suitable for genome or transcriptome alignment, but not both. Pufferfish fills the gap by allowing fast and accurate mapping to a collection of genomes and annotated transcripts at the same time, achieveing the sensitivity of transcriptome-based aligners and the robustness of genome-based aligners. While we have not built a spliced-aligner on top of Pufferfish yet, meaning that it cannot be easily used to derive new splicing junctions, we can already make use of it's ability to align reads against the transcriptomic and genomic tagets simultaneously beyond it's ability to produce improved accuracy alignments. One immediate outcome of having short reads mapped to both genome and transcriptome is in RNA-seq quality control. If we just look at the transcriptome mapping outcome, we could simply throw all the non-mapped reads out, ignoring the fact that not being mapped at all is a different observation than being mapped to an intron or an intergenic region. A large fraction of reads mapping to introns could be the evidence that the RNA-seq experiment failed to provide the required quality (poor selection) or it could be evidence of the biological signal of intron retention.

The utility of performing joint genome and transcriptome alignment is demonstrated in recent work [152](a paper that is the teamwork of almost all the members of the Combine-Lab). We demonstrate the improvement in accuracy that results from aligning reads to both the genome and transcriptome simultaneously. As another use of the Pufferfish index for efficient alignment, we focus on applying Pufferfish, along with a probabilistic model, to metagenomic data to improve accuracy, space and time requirements for abundance estimation. The tool implementing these ideas is under active development in our new tool, "Cedar", which also takes advantage of certain domain-specific characteristics of the underlying data.

2 Reference-free Indexing

We also discussed different methods and indices for sequence search in chapter 5 and introduced our tool Mantis. Mantis is an exact indexing data structure (no false positives) for querying massive RNA-sequencing databases. It enables sequence search queries that are efficient in space and time compared to other tools. It uses the counting quotient filter to index k-mers and the representation introduced in Rainbowfish and explained in chapter 4 to represent the samples containing the k-mer. As shown in chapter 5, Mantis was orders of magnitude faster than existing state-of-the-art tools, while also being somewhat smaller. We were able to scale Mantis to $10,000$ raw sequencing samples through modifications introduced in MST-based Mantis. However, even then, challenges to scalability remained. Comparing the Classic Mantis with the MST-based Mantis representation, we observed a turning

point in the scalability bottleneck from the color information to the k-mer filter representation (counting quotient filter) as the indexed datasets grew larger. Further, the necessity of having the (compressed) color bit matrix available as a prerequisite to construct the MST over the color bit-vectors was another obstacle to achieving improved scalability.

In chapter 6, we explored the details of the algorithms to merge each of the Classic Mantis and the MST-based Mantis data structures. We devised an efficient methodology for direct merge of the MSTs, which gives us the opportunity to discard the construction/merging step for the large color matrix, and thus saves an immense amount of intermediate disk space. The only challenge for scaling that construction process is keeping the memory usage low. To address this concern, we propose a scheme for partitioning the counting quotient filter into a set of disjoint partitioned CQFs and substitute the loading of one massive counting quotient filter with the loading of many small partitioned CQFs, one at a time. Through an intelligent multi-pass algorithm that avoids holding all data for the input and output structures in memory at once, we address the scalability challenge.

In fact, we show that the memory for merging the input partitioned CQFs and also using the output partitioned CQF during the MST merge is constant and bounded to a multiple of the size of a partitioned CQF block. Given the ability to efficiently merge both classic and MST-based Mantis data structures, we demonstrate that one way to dynamize the structure is using the framework of log-structured merge trees. Achieving a scalable and updatable cosntruction process for MST-based Mantis allowed us to extend the index to over $15k$ samples by adding

experiments incrementally. The future work in the line of Mantis indexing is to (1) achieve aggressive memory reduction for construction query by giving up the exactness and (2) set up a distributed Mantis index over a network of low-cost commodity systems on a large number of samples. A distributed mantis index could conceivably be built over all publicly-available RNA-seq data, and would provide a sequence-queryable and updatable online index for searching over the vast public repository of available RNA-seq data.

Chapter 7: List of Projects

In this chapter I list all the completed and ongoing projects that I have participated in throughout my PhD and list the venue if the work has been published.

List of Publications:

1. *On the distribution of lexical features at multiple levels of analysis*, **F Almodaresi**, L Ungar, V Kulkarni, M Zakeri, S Giorgi, HA Schwartz, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) 2 2017

2. *Rainbowfish: a succinct colored de Bruijn graph representation*, **F Almodaresi**, P Pandey, R Patro, 17th International Workshop on Algorithms in Bioinformatics (WABI 2017), 36, 2017

3. *Improved data-driven likelihood factorizations for transcript abundance estimation*, M Zakeri, A Srivastava, **F Almodaresi**, R Patro, Bioinformatics 33 (14), i142-i151 12 2017

4. *A space and time-efficient index for the compacted colored de Bruijn graph*, **F Almodaresi**, H Sarkar, A Srivastava, R Patro, Bioinformatics 34 (13), i169-i177 17 2018

5. *Mantis: A fast, small, and exact large-scale sequence-search index*, P

Pandey, **F Almodaresi**, MA Bender, M Ferdman, R Johnson, R Patro, Cell systems 7 (2), 201-207. e4 39 2018

6. *Grouper: graph-based clustering and annotation for improved de novo transcriptome analysis*, L Malik, **F Almodaresi**, R Patro, Bioinformatics 34 (19), 3265-3272 6 2018

7. *Alignment and mapping methodology influence transcript abundance estimation*, A Srivastava, L Malik, H Sarkar, M Zakeri, **F Almodaresi**, C Soneson, MI Love, C Kingsford, R Patro, BioRxiv, 657874 4 2019

8. *An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search*, **F Almodaresi**, P Pandey, M Ferdman, R Johnson, R Patro, International Conference on Research in Computational Molecular Biology, 1-18 6 2019

9. *An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search*, **F Almodaresi**, P Pandey, M Ferdman, R Johnson, R Patro, Journal of Computational Biology 27 (4), 485-499 1 2020

Current (Unpublished) Work:

1. *AGAMEMNON: an Accurate metaGenomics And MEtatranscriptoMics quaNtificatiON analysis suite*, G Skoufos, **F Almodaresi**, M Zakeri, JN Paulson, R Patro, AG Hatzigeorgiou, and IS Vlachos.

2. *Puffaligner : A Fast, Efficient, and Accurate Aligner Based on the Pufferfish Index*, **F Almodaresi**, M Zakeri, R Patro.

3. *An incrementally-updatable and scalable system for large-scale sequence search using LSM-trees*, **F Almodaresi**, J Khan, S Madaminov, P Pandey, M Ferdman, R Johnson, and R Patro.

4. *Cedar*, A metagenomic abundance estimation pipeline, **F Almodaresi**, M zakeri, R Patro.

Bibliography

[1] Bahar Alipanahi, Alan Kuhnle, and Christina Boucher. Recoloring the Colored de Bruijn Graph. In *International Symposium on String Processing and Information Retrieval*, pages 1–11. Springer, 2018.

[2] Bahar Alipanahi, Martin D Muggli, Musa Jundi, Noelle Noyes, and Christina Boucher. Resistome SNP calling via read colored de Bruijn graphs. *bioRxiv*, page 156174, 2018.

[3] Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, Gozde Aksay, Francesca Antonacci, Fereydoun Hormozdiari, Jacob O Kitzman, Carl Baker, Maika Malig, Onur Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061, 2009.

[4] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: a succinct colored de bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

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

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

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

[8] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable and exact representation of high-dimensional color information enabled via de bruijn graph search. In *International Conference on Research in Computational Molecular Biology*, pages 1–18. Springer, 2019.

[9] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable, and exact representation of high-dimensional color information enabled using de bruijn graph search. *Journal of Computational Biology*, 27(4):485–499, 2020.

[10] Ernst Althaus, Stefan Funke, Sariel Har-Peled, Jochen Könemann, Edgar A. Ramos, and Martin Skutella. Approximating k-hop minimum-spanning trees. *Operations Research Letters*, 33(2):115 – 120, 2005. ISSN 0167-6377. doi: https://doi.org/10.1016/j.orl.2004.05.005. URL `http://www.sciencedirect.com/science/article/pii/S0167637704000719`.

[11] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215 (3):403–410, 1990.

[12] Pavel V Baranov, Clark M Henderson, Christine B Anderson, Raymond F Gesteland, John F Atkins, and Michael T Howard. Programmed ribosomal frameshifting in decoding the sars-cov genome. *Virology*, 332(2):498–510, 2005.

[13] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, and Marco Previtali. Fully dynamic de bruijn graphs. In *International Symposium on String Processing and Information Retrieval*, pages 145–152. Springer, 2016.

[14] Timo Beller and Enno Ohlebusch. A representation of a compressed de bruijn graph for pan-genome analysis that enables search. *Algorithms for Molecular Biology*, 11(1), July 2016.

[15] Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019.

[16] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[17] A Bookstein and ST Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[18] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proceedings of the International Workshop on Algorithms in Bioinformatics*, pages 225–235. Springer, 2012.

[19] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *International Workshop on Algorithms in Bioinformatics*, pages 225–235. Springer, 2012.

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

[21] Phelim Bradley, Henk C Den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152–159, 2019.

[22] Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525–527, 2016.

[23] Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525, 2016.

[24] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[25] Mathilde Causse, Nelly Desplat, Laura Pascual, Marie-Christine Le Paslier, Christopher Sauvage, Guillaume Bauchet, Aurélie Bérard, Rémi Bounon, Maria Tchoumakov, Dominique Brunel, et al. Whole genome resequencing in tomato reveals variation associated with introgression and breeding events. *BMC genomics*, 14(1):791, 2013.

[26] Zheng Chang, Guojun Li, Juntao Liu, Yu Zhang, Cody Ashby, Deli Liu, Carole L Cramer, and Xiuzhen Huang. Bridger: a new framework for de novo transcriptome assembly using rna-seq data. *Genome Biol*, 16(1):30, 2015.

[27] Shifu Chen, Yanqing Zhou, Yaru Chen, and Jia Gu. fastp: an ultra-fast all-in-one fastq preprocessor. *Bioinformatics*, 34(17):i884–i890, 2018.

[28] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. In *International Workshop on Algorithms in Bioinformatics*, pages 236–248. Springer, 2012.

[29] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8 (1):22, 2013.

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

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

[32] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[33] MetaSUB International Consortium et al. The metagenomics and metadesign of the subways and urban biomes (metasub) international consortium inaugural meeting report, 2016.

[34] Jake R Conway, Alexander Lex, and Nils Gehlenborg. UpSetR: an R package for the visualization of intersecting sets and their properties. *Bioinformatics*, 33(18):2938–2940, 2017.

[35] Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[36] Victoria Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, Travis Gagie, and John Hancock. Practical dynamic de bruijn graphs. *Bioinformatics*, 2018.

[37] Matei David, Misko Dzamba, Dan Lister, Lucian Ilie, and Michael Brudno. Shrimp2: sensitive yet practical short read mapping. *Bioinformatics*, 27(7): 1011–1012, 2011.

[38] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[39] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. Star: ultrafast universal rna-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[40] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[41] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. Gatb: Genome assembly & analysis tool box. *Bioinformatics*, 30(20):2959–2961, 2014.

[42] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[43] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[44] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[45] Scott Federhen. The ncbi taxonomy database. *Nucleic acids research*, 40(D1): D136–D143, 2012.

[46] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[47] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[48] Adam Frankish, Mark Diekhans, Anne-Maud Ferreira, Rory Johnson, Irwin Jungreis, Jane Loveland, Jonathan M Mudge, Cristina Sisu, James Wright, Joel Armstrong, et al. Gencode reference annotation for the human and mouse genomes. *Nucleic acids research*, 47(D1):D766–D773, 2019.

[49] Alyssa C Frazee, Andrew E Jaffe, Ben Langmead, and Jeffrey T Leek. Polyester: simulating rna-seq datasets with differential transcript expression. *Bioinformatics*, 31(17):2778–2784, 2015.

[50] Dirk Gevers, Rob Knight, Joseph F Petrosino, Katherine Huang, Amy L McGuire, Bruce W Birren, Karen E Nelson, Owen White, Barbara A Methé, and Curtis Huttenhower. The human microbiome project: a community resource for the healthy human microbiome. *PLoS Biol*, 10(8):e1001377, 2012.

[51] Simon Gog. Succinct data structure library. `https://github.com/simongog/sdsl-lite`, 2017. [online; accessed 01-Feb-2017].

[52] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[53] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[54] Manfred G Grabherr, Brian J Haas, Moran Yassour, Joshua Z Levin, Dawn A Thompson, Ido Amit, Xian Adiconis, Lin Fan, Raktima Raychowdhury, Qiandong Zeng, et al. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature biotechnology*, 29(7):644–652, 2011.

[55] Manfred G Grabherr, Brian J Haas, Moran Yassour, Joshua Z Levin, Dawn A Thompson, Ido Amit, Xian Adiconis, Lin Fan, Raktima Raychowdhury, Qiandong Zeng, et al. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature biotechnology*, 29(7):644–652, 2011.

[56] Brian J Haas, Alexie Papanicolaou, Moran Yassour, Manfred Grabherr, Philip D Blood, Joshua Bowden, Matthew Brian Couger, David Eccles, Bo Li, Matthias Lieber, et al. De novo transcript sequence reconstruction from rna-seq: reference generation and analysis with trinity. *Nature protocols*, 8(8), 2013.

[57] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8):576–577, 2010.

[58] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsfast: a cache-oblivious algorithm for short-read mapping. *Nature methods*, 7(8):576, 2010.

[59] Robert S Harris and Paul Medvedev. Improved representation of sequence bloom trees. *Bioinformatics*, 36(3):721–727, 2020.

[60] J. Harrow, A. Frankish, J. M. Gonzalez, E. Tapanari, M. Diekhans, F. Kokocinski, B. L. Aken, D. Barrell, A. Zadissa, S. Searle, I. Barnes, A. Bignell, V. Boychenko, T. Hunt, M. Kay, G. Mukherjee, J. Rajan, G. Despacio-Reyes, G. Saunders, C. Steward, R. Harte, M. Lin, C. Howald, A. Tanzer, T. Derrien, J. Chrast, N. Walters, S. Balasubramanian, B. Pei, M. Tress, J. M. Rodriguez, I. Ezkurdia, J. van Baren, M. Brent, D. Haussler, M. Kellis, A. Valencia, A. Reymond, M. Gerstein, R. Guigo, and T. J. Hubbard. GENCODE: The reference human genome annotation for the ENCODE project. *Genome Research*, 22(9):1760–1774, sep 2012. doi: 10.1101/gr.135350.111. URL `https://doi.org/10.1101%2Fgr.135350.111`.

[61] Mahdi Heydari, Giles Miclotte, Yves Van de Peer, and Jan Fostier. Browniealigner: accurate alignment of illumina sequencing data to de bruijn graphs. *BMC bioinformatics*, 19(1):311, 2018.

[62] Guillaume Holley and Páll Melsted. Bifrost–highly parallel construction and indexing of colored and compacted de bruijn graphs. *BioRxiv*, page 695338, 2019.

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

[64] Manuel Holtgrewe. Mason: a read simulator for second generation sequencing data. 2010.

[65] Ellis Horowitz and Sartaj Sahni. *Fundamentals of computer algorithms*. Computer Science Press, 1978.

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

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

[68] Iqbal Zamin, Caccamo Mario, Turner Isaac, Flicek Paul, and McVean Gil. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226—232, January 2012. doi: http://dx.doi.org/10.1038/ng.102810.1038/ng.1028.

[69] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

[70] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In *International Conference on Research in Computational Molecular Biology*, pages 66–81. Springer, 2017.

[71] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. *Journal of Computational Biology*, 25(7):766–779, July 2018. doi: 10.1089/cmb.2018.0036. URL `https://doi.org/10.1089/cmb.2018.0036`.

[72] Samir Khuller, Balaji Raghavachari, and Neal E. Young. Balancing minimum spanning and shortest path trees. *CoRR*, cs.DS/0205045, 2002. URL `http://arxiv.org/abs/cs.DS/0205045`.

[73] Daehwan Kim, Ben Langmead, and Steven L Salzberg. Hisat: a fast spliced aligner with low memory requirements. *Nature methods*, 12(4):357–360, 2015.

[74] Daehwan Kim, Ben Langmead, and Steven L Salzberg. HISAT: a fast spliced aligner with low memory requirements. *Nature methods*, 12(4):357–360, 2015.

[75] Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. Graph-based genome alignment and genotyping with hisat2 and hisat-genotype. *Nature biotechnology*, 37(8):907–915, 2019.

[76] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic acids research*, 40(D1): D54–D56, 2011.

[77] Ben Langmead. Aligning short sequencing reads with bowtie. *Current protocols in bioinformatics*, 32(1):11–7, 2010.

[78] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357, 2012.

[79] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[80] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[81] Rasko Leinonen, Hideaki Sugawara, Martin Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. *Nucleic acids research*, 39(suppl_1):D19–D21, 2010.

[82] Bo Li and Colin N Dewey. RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome. *BMC Bioinformatics*, 12 (1):323, 2011.

[83] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[84] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[85] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[86] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[87] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[88] Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18 (11):1851–1858, 2008.

[89] Yang Liao, Gordon K Smyth, and Wei Shi. The subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic acids research*, 41(10): e108–e108, 2013.

[90] Yang Liao, Gordon K Smyth, and Wei Shi. The subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic acids research*, 41(10): e108–e108, 2013.

[91] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. Read mapping on de bruijn graphs. *BMC bioinformatics*, 17(1):237, 2016.

[92] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. Read mapping on de bruijn graphs. *BMC bioinformatics*, 17(1):237, 2016.

[93] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *arXiv preprint arXiv:1702.03154*, 2017.

[94] Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. debga: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32 (21):3224–3232, 2016.

[95] Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. debga: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32 (21):3224–3232, 2016.

[96] Bo Liu, Yadong Liu, Tianyi Zang, and Yadong Wang. desalt: fast and accurate long transcriptomic read alignment with de bruijn graph-based index. *bioRxiv*, page 612176, 2019.

[97] Juntao Liu, Guojun Li, Zheng Chang, Ting Yu, Bingqiang Liu, Rick McMullen, Pengyin Chen, and Xiuzhen Huang. Binpacker: Packing-based de novo transcriptome assembly from rna-seq data. *PLOS Comput Biol*, 12(2): e1004772, 2016.

[98] John Lonsdale, Jeffrey Thomas, Mike Salvatore, Rebecca Phillips, Edmund Lo, Saboor Shad, Richard Hasz, Gary Walters, Fernando Garcia, Nancy Young, et al. The genotype-tissue expression (gtex) project. *Nature genetics*, 45(6):580, 2013.

[99] Jennifer Lu, Florian P Breitwieser, Peter Thielen, and Steven L Salzberg. Bracken: estimating species abundance in metagenomics data. *PeerJ Computer Science*, 3:e104, 2017.

[100] Liping Ma, Bing Li, and Tong Zhang. Abundant rifampin resistance genes and significant correlations of antibiotic resistance genes and plasmids in various environments revealed by metagenomic analysis. *Applied microbiology and biotechnology*, 98(11):5195–5204, 2014.

[101] Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a burrows-wheeler transform to enable mapping and genome inference. In *International Workshop on Algorithms in Bioinformatics*, pages 222–233. Springer, 2016.

[102] Prabhu Manyem and Matthias F. M. Stallmann. Some approximation results in multicasting. Technical report, Raleigh, NC, USA, 1996.

[103] Madhav V. Marathe, R. Ravi, Ravi Sundaram, S. S. Ravi, Daniel J. Rosenkrantz, and Harry B. Hunt III. Bicriteria network design problems. *CoRR*, cs.CC/9809103, 1998. URL `http://arxiv.org/abs/cs.CC/9809103`.

[104] Alexa B. R. McIntyre, Rachid Ounit, Ebrahim Afshinnekoo, Robert J. Prill, Elizabeth Hénaff, Noah Alexander, Samuel S. Minot, David Danko, Jonathan Foox, Sofia Ahsanuddin, Scott Tighe, Nur A. Hasan, Poorani Subramanian, Kelly Moffat, Shawn Levy, Stefano Lonardi, Nick Greenfield, Rita R. Colwell, Gail L. Rosen, and Christopher E. Mason. Comprehensive benchmarking and

ensemble approaches for metagenomic classifiers. *Genome Biology*, 18(1), sep 2017.

[105] Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, page btw609, 2016.

[106] Ilya Minkin, Anand Patel, Mikhail Kolmogorov, Nikolay Vyahhi, and Son Pham. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In *International Workshop on Algorithms in Bioinformatics*, pages 215–229. Springer, 2013.

[107] Narjes S. Movahedi, Elmirasadat Forouzmand, and Hamidreza Chitsaz. De novo co-assembly of bacterial genomes from multiple single cells. In *2012 IEEE International Conference on Bioinformatics and Biomedicine*. IEEE, October 2012.

[108] Martin D. Muggli. Vari. `https://github.com/cosmo-team/cosmo/tree/VARI`, February 2017. Viewed Feb 3, 2017.

[109] Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul Morley, Keith Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct Colored de Bruijn Graphs. 2017.

[110] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul Morley, Keith Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, page btx067, 2017.

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

[112] Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.

[113] Harun Mustafa, Ingo Schilken, Mikhail Karasikov, Carsten Eickhoff, Gunnar Rätsch, and André Kahles. Dynamic compression schemes for graph coloring. *Bioinformatics*, page bty632, 2018. doi: 10.1093/bioinformatics/bty632. URL `http://dx.doi.org/10.1093/bioinformatics/bty632`.

[114] NIH. SRA. `https://www.ncbi.nlm.nih.gov/sra`, 2017. [online; accessed 06-Nov-2017].

[115] Noelle R Noyes, Xiang Yang, Lyndsey M Linke, Roberta J Magnuson, Adam Dettenwanger, Shaun Cook, Ifigenia Geornaras, Dale E Woerner, Sheryl P Gow, Tim A McAllister, et al. Resistome diversity in cattle and the environment decreases during beef production. 5:e13195, 2016.

[116] Nuala A O'Leary, Mathew W Wright, J Rodney Brister, Stacy Ciufo, Diana Haddad, Rich McVeigh, Bhanu Rajput, Barbara Robbertse, Brian Smith-White, Danso Ako-Adjei, et al. Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. page gkv1189, 2015.

[117] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):132, 2016.

[118] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 273–282. ACM, 2014.

[119] Rachid Ounit, Steve Wanamaker, Timothy J Close, and Stefano Lonardi. Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC genomics*, 16(1):1, 2015.

[120] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[121] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: An exact and approximate k-mer counting system. *Bioinformatics*, page btx636, 2017. doi: 10.1093/bioinformatics/btx636. URL +http://dx.doi.org/10.1093/bioinformatics/btx636.

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

[123] Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index. *Cell Systems*, 7(2):201–207.e4, Aug 2018. doi: 10.1016/j.cels.2018.05.021. URL https://doi.org/10.1016/j.cels.2018.05.021.

[124] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems*, 7(2):201–207, 2018.

[125] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome research*, 27(5):665–676, 2017.

[126] Rob Patro, Stephen M Mount, and Carl Kingsford. Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nature biotechnology*, 32(5):462–464, 2014.

[127] Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*, 14(4):417, 2017.

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

[129] Pavel A Pevzner and Haixu Tang. Fragment assembly with double-barreled data. *Bioinformatics*, 17(suppl_1):S225–S233, 2001.

[130] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[131] Günther R. Raidl. *Exact and Heuristic Approaches for Solving the Bounded Diameter Minimum Spanning Tree Problem*. PhD thesis, 2008.

[132] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[133] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(7):701–710, July 1994. ISSN 0162-8828. doi: 10.1109/34.297950. URL `https://doi.org/10.1109/34.297950`.

[134] Mark Reppell and John Novembre. Using pseudoalignment and base quality to accurately quantify microbial community composition. *PLoS computational biology*, 14(4):e1006096, 2018.

[135] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[136] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP91)*, pages 1–15, October 1991.

[137] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10 (1):26–52, February 1992. doi: 10.1145/146941.146943. , Volume 10, Number 1, February.

[138] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. In *International Workshop on Algorithms in Bioinformatics*, pages 364–376. Springer, 2013.

[139] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, 9(1):2, 2014.

[140] Hirak Sarkar, Mohsen Zakeri, Laraib Malik, and Rob Patro. Towards selective-alignment: Producing accurate and sensitive alignments using quasi-mapping. *bioRxiv*, page 138800, 2017.

[141] Hirak Sarkar, Mohsen Zakeri, Laraib Malik, and Rob Patro. Towards selective-alignment: Bridging the accuracy gap between alignment-based and alignment-free transcript quantification. In *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 27–36, Washington DC, USA, 2018. ACM. URL `http://doi.acm.org/10.1145/3233547.3233589`.

[142] Patrick S Schnable, Doreen Ware, Robert S Fulton, Joshua C Stein, Fusheng Wei, Shiran Pasternak, Chengzhi Liang, Jianwei Zhang, Lucinda Fulton, Tina A Graves, et al. The b73 maize genome: complexity, diversity, and dynamics. *science*, 326(5956):1112–1115, 2009.

[143] Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome biology*, 10(9): R98, 2009.

[144] Marcel H Schulz, Daniel R Zerbino, Martin Vingron, and Ewan Birney. Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–1092, 2012.

[145] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[146] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[147] Jouni Sirén. Indexing variation graphs. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, January 2017.

[148] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 2016.

[149] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300–302, 2016.

[150] Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. In *International Conference on Research in Computational Molecular Biology*, pages 257–271. Springer, 2017.

[151] Martin Šošić and Mile Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[152] Avi Srivastava, Laraib Malik, Hirak Sarkar, Mohsen Zakeri, Fatemeh Almodaresi, Charlotte Soneson, Michael I Love, Carl Kingsford, and Rob Patro. Alignment and mapping methodology influence transcript abundance estimation. *BioRxiv*, page 657874, 2019.

[153] Chen Sun, Robert S Harris, Rayan Chikhi, and Paul Medvedev. Allsome sequence bloom trees. In *International Conference on Research in Computational Molecular Biology*, pages 272–286. Springer, 2017.

[154] Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19(1):45, 2018.

[155] David Swarbreck, Christopher Wilks, Philippe Lamesch, Tanya Z Berardini, Margarita Garcia-Hernandez, Hartmut Foerster, Donghui Li, Tom Meyer, Robert Muller, Larry Ploetz, et al. The arabidopsis information resource (tair): gene structure and function annotation. *Nucleic acids research*, 36 (suppl 1):D1009–D1014, 2008.

[156] Tsuyoshi Tanaka, Baltazar A Antonio, Shoshi Kikuchi, Takashi Matsumoto, Yoshiaki Nagamura, Hisataka Numa, Hiroaki Sakai, Jianzhong Wu, Takeshi Itoh, Takuji Sasaki, et al. The rice annotation project database (rap-db): 2008 update. *Nucleic Acids Research*, 36(Supp 1):D1028–D1033, 2008.

[157] Xiaolu Tang, Changcheng Wu, Xiang Li, Yuhe Song, Xinmin Yao, Xinkai Wu, Yuange Duan, Hong Zhang, Yirong Wang, Zhaohui Qian, et al. On the origin and continuing evolution of sars-cov-2. *National Science Review*, 2020.

[158] Isaac Turner, Kiran V Garimella, Zamin Iqbal, and Gil McVean. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics*, 34(15):2556–2565, 2018. doi: 10.1093/bioinformatics/bty157. URL `http://dx.doi.org/10.1093/bioinformatics/bty157`.

[159] Yixuan Wang, Yuyi Wang, Yan Chen, and Qingsong Qin. Unique epidemiological and clinical features of the emerging 2019 novel coronavirus pneumonia (covid-19) implicate special control measures. *Journal of medical virology*, 92 (6):568–576, 2020.

[160] Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1, 2014.

[161] Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1–12, 2014.

[162] Fan Wu, Su Zhao, Bin Yu, Yan-Mei Chen, Wen Wang, Zhi-Gang Song, Yi Hu, Zhao-Wu Tao, Jun-Hua Tian, Yuan-Yuan Pei, et al. A new coronavirus associated with human respiratory disease in china. *Nature*, 579(7798):265–269, 2020.

[163] Y William Yu, Noah M Daniels, David Christian Danko, and Bonnie Berger. Entropy-scaling search of massive biological data. *Cell systems*, 1(2):130–140, 2015.

[164] Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. SeqOthello: querying RNA-seq experiments at scale. *Genome Biology*, 19(1):167, Oct 2018. ISSN 1474-760X. doi: 10.1186/s13059-018-1535-9. URL `https://doi.org/10.1186/s13059-018-1535-9`.

[165] Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. Seqothello: querying rna-seq experiments at scale. *Genome biology*, 19(1):167, 2018.

[166] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[167] Tao Zhang, Qunfu Wu, and Zhigang Zhang. Probable pangolin origin of sars-cov-2 associated with the covid-19 outbreak. *Current Biology*, 2020.

[168] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.