Annual Review of Biomedical Data Science

Sketching and Sublinear Data Structures in Genomics

Guillaume Marçais,[1,*] Brad Solomon,[2,*] Rob Patro,[3] and Carl Kingsford[1]

[1]Computational Biology Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, USA; email: gmarcais@cs.cmu.edu, carlk@cs.cmu.edu

[2]Department of Computer Science, Whiting School of Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA; email: brad.sol@gmail.com

[3]Department of Computer Science, Stony Brook University, Stony Brook, New York 11794, USA; email: rob.patro@cs.stonybrook.edu

Annu. Rev. Biomed. Data Sci. 2019. 2:93–118

First published as a Review in Advance on April 8, 2019

The *Annual Review of Biomedical Data Science* is online at biodatasci.annualreviews.org

https://doi.org/10.1146/annurev-biodatasci-072018-021156

*These authors contributed equally to this article

Keywords

genomics, sketching, succinct data structures

Abstract

Large-scale genomics demands computational methods that scale sublinearly with the growth of data. We review several data structures and sketching techniques that have been used in genomic analysis methods. Specifically, we focus on four key ideas that take different approaches to achieve sublinear space usage and processing time: compressed full-text indices, approximate membership query data structures, locality-sensitive hashing, and minimizers schemes. We describe these techniques at a high level and give several representative applications of each.

1. INTRODUCTION

Genomics sequencing capacity and throughput have been growing rapidly (1). Short read sequencers such as Illumina or long read sequencers such as Pacific Biosciences' SMRT (single-molecule real-time) sequencing or Oxford Nanopore Technologies' MinION and PromethION can sequence tens to hundreds of billions of nucleotides per day, at an ever-decreasing cost per base. Driven by falling costs and increasing availability, the number of applications using sequencing data has been increasing rapidly. We have not only a human reference genome but also the genomes of thousands of individuals from across the world (2, 3). Large sequencing projects have also been undertaken to profile the genomes and expression profiles of many cancerous cells (often paired with sequencing from healthy cells of the same individual) (4). In the realm of gene expression profiling, RNA sequencing (RNA-seq) (5) has replaced microarrays in many cases, with twice as many RNA-seq submissions to GEO (Gene Expression Omnibus) in 2018 (6), and the amount of RNA-seq data produced is also growing rapidly.

Public repositories of genomics data (7–9) have seen an exponential growth for over two decades: The National Insitutes of Health's Sequence Read Archive and the European Nucleotide Archive contain close to ten quadrillion bases and are expected to double in less than two years (7, 9). As we enter the age of personalized medicine, hospitals and insurance companies will also begin to routinely sequence patients, and they will therefore face the challenges of processing and storing vast amounts of sensitive genomics data. In addition to storage, the data must be efficiently computationally manipulatable for further analysis to achieve the promised benefit of personal medicine. In the medical setting, the timely processing and querying of data are even more crucial than in a research lab environment.

The resulting large amount of sequencing data has raised new computational challenges. Crucially, the approaches used to store, index, and search the data must scale sublinearly. Even a few thousand experimental samples could quickly exceed the memory and disk capacities of modern servers. To be practical, the runtimes and space used by algorithms for processing the data must grow more slowly than the raw data being processed (10, 11).

Many computational techniques have been developed to meet this challenge. These techniques enable faster and more space-efficient processing of large collections of sequence data by representing the data's essential features, which depend on the application, using space-efficient data structures. Some of these data structures were developed for general string data and have been adapted and applied to biological applications. Others were specifically developed for genomics challenges.

Four broad classes of ideas have been crucial to these efforts. The first consists of compact, compressed indices that allow for efficient exact string matching within a large collection of sequences. Techniques such as the FM-index (12) can index strings using space close to the information-theoretic minimum, while still supporting exact search. The second class of ideas comprises dimensionality reduction techniques, most relevantly locality-sensitive hashing (LSH) schemes. These methods allow distances between data points to be computed efficiently but with some error by representing each datum by a probabilistic sketch that can be compared with the sketches of other data or queries. Such sketches, for example, support efficient nearest-neighbor queries. The third is the broad class of data structures that support approximate membership queries (AMQs). In many cases, sequencing data can be usefully represented as a set (e.g., of substrings), and AMQ data structures allow that set to be stored in small space, at the cost usually of both some errors and some limits on the types of operations that can be performed. A fourth class of ideas, minimizers and local schemes, is more specific to sequence analysis and is also used to provide reduced representations of strings for various applications.

Although the jargon "sketch" is usually applied in the literature to only ideas of the second class, each of these classes of ideas represents a way of sketching sequence collections, either with or without error, using small space. The techniques support different operations with different trade-offs, so each is suited to its own set of applications. This is a theme of much research in bioinformatics: to find or design the appropriate, optimally sized representation that will support a particular application.

We review these four classes of ideas and some of the specific data structures that employ them. We give high-level descriptions of the various computational techniques and discuss example applications that use them to make biological sequence analysis more practical (see **Table 1**).

Table 1 Software and related applications

Software (references)	Application(s)	Section(s)
Suffix trees and arrays software		
MUMmer (13, 14)	Genome-to-genome alignment	3.3.1
BWT and FM-index software		
Bowtie (15, 16)	Short read alignment	3.3.1
BWA (17, 18)	Short and long read alignment	3.3.1
BLASR (19)	Long read alignment	3.3.1
HISAT (20)	Gapped read alignment using a hierarchy of FM-indices	3.3.1
SOAP2 (21)	Short read alignment	3.3.1
SGA (22)	Genome assembly with an FM-index	3.3.2
DBGFM (23, 24)	De Bruijn graph as an FM-index	3.3.2, 6.3.4
Compressive genomics software		
CaBLAST (25, 26)	Sequence alignment with reference compression	3.3.1
CORA (27)	Alignment acceleration with query compression	3.3.1
Approximate membership query software		
Jellyfish (28)	*k*-mer counting; BF filters singleton *k*-mers	5.6.1
BFCounter (29)	*k*-mer counting; BF filters singleton *k*-mers	5.6.1
SBT (30, 31)	Hierarchy of BFs for sequence search	5.4, 5.6.2
SBT-ALSO (32)	Hierarchy of BFs for sequence search	5.6.2
Mantis (33)	Counting quotient filter for sequence search	5.6.2
Counting sketch software		
khmer (34)	minCount sketch for *k*-mer distribution	4.4.1
ntCard (35)	ntHash sketch for *k*-mer distribution	4.3, 4.4.1
LSH software		
LSH-ALL-PAIRS (36)	All-versus-all alignment using Hamming distance	4.4.2
MHAP (37)	Read overlap using Jaccard index	4.4.2
MashMap (38)	Long read alignment using Jaccard index and minimizers method	4.4.2, 6.3.5
LSH-Div (39)	Metagenomics abundance estimation using Hamming distance	4.4.3
Mash (40)	Metagenomics abundance estimation using Jaccard index	4.1, 4.4.3
sourmash (41)	Metagenomics abundance estimation using Jaccard index	4.1, 4.4.3
Minimizers methods software		
minimap (42, 43)	Long read alignment	6.3.2
MSPKmerCounter (44)	*k*-mer counter using super *k*-mers	6.3.3
KMC (45)	*k*-mer counter using super *k*-mers	6.3.3
SparseAssembler (46)	Sparse de Bruijn graph for genome assembly	6.3.5
SamSAMi (47)	Sparse suffix array	6.3.5

Abbreviations: BF, Bloom filter; BWT, Burrows–Wheeler transform; LSH, locality-sensitive hashing.

Naturally, we are forced to omit some techniques and applications due to space constraints—most obviously pure sequence compression techniques that aim only at reduced storage, and techniques relating specifically to other kinds of data such as imaging or molecular interaction networks. Many of the techniques we discuss here can be used in both streaming algorithms, where the data can be read only once, and off-line settings. The off-line setting is currently the common one in genomics and is the focus of this review. We have selected the techniques we cover with an eye to their importance and their representativeness. Most of the techniques have applications (often their original ones) beyond genomics, attesting to the general applicability of these methods.

Our aims are to broaden the appreciation of these sketching and data structuring techniques and to provide a starting point for those interested in applying or improving on them. Each of the four classes of ideas is the basis for multiple lines of active research: to develop new techniques, to improve the efficiency of existing ones, and to find ways of formulating new sequence analysis applications so that they can be profitably exploited.

2. BACKGROUND

2.1. Sequencing

Any genomics analysis considered here starts with either DNA sequencing or RNA-seq. In DNA sequencing, the actual genomic material of the cell (chromosomes or plasmids) is sequenced (i.e., read by a machine) to obtain sequencing fragments sometimes called reads. Because RNA cannot be sequenced directly with some technologies, in RNA-seq (5) the messenger RNA (mRNA) of a cell is captured and used for complementary DNA (cDNA) synthesis, which is then sequenced. Two main categories of sequencing technologies dominate currently: short read sequencing and long read sequencing:

- Short read sequencing, also called next-generation sequencing (NGS) or second-generation sequencing, generates relatively short reads (between 30 and 400 bp) with a low error rate (fewer than 1% of the bases are erroneous, and most errors are substitutions). Common NGS technologies include Ion Torrent^{TM} semiconductor sequencing (48), SOLiD sequencing (49), and Illumina sequencing (50). These technologies are used for both DNA sequencing and RNA-seq. Illumina sequencing is by far the most commonly used technology in this class.
- In contrast to short read data, long read sequencing (also called third-generation sequencing) is designed to produce long, continuous strands of sequence information at the expense of both throughput and accuracy. The read length varies greatly from 1,000 to 100,000 bp, with an average read length around 30,000 bp for the most recent chemistry, but with a per base error rate of 10–15% (51, 52). The two primary technologies in this class are Nanopore sequencing (53), which uses electric density across a nucleotide-sized gap to measure the composition of the DNA being passed through it, and SMRT sequencing (54), which uses fluorescent signals during the incorporation of a nucleotide to map the composition. Long read sequencing has been used for DNA sequencing and more recently has been adapted to RNA-seq.

For this review, we can view each technology as providing a large set of strings over a small alphabet (typically {A, C, G, T}), where each string is sampled randomly (according to some distribution) from some unknown (set of) longer strings. Since each sequenced fragment is short, in most applications many copies of the underlying molecule are sequenced at the same time, providing overlapping and partially redundant views of the molecule. In a genome sequencing project, 30× to 100× coverage may be expected, meaning that each base is expected to be sequenced 30 to 100 times. This leads to even larger data sizes that motivate the development of the techniques we discuss below. An effort to sequence a new species with approximately the same genome size as a human may lead to a collection of 400 million fragments with a total of 90 gigabytes or more of raw sequence data. This is for one individual. When many individuals of a species are sequenced, the raw data can grow proportionally.

2.2. Biological Applications

We highlight below some motivating problems that have driven the development of the new computational techniques reviewed here. More in-depth discussion of these applications appears after we introduce each technique.

2.2.1. Genome assembly. The genome assembly task is to reconstruct a species' genome from the sequenced fragments of its genome. The idea is to find overlaps between fragments sampled from nearby places in the true, unknown genome. The genome assembly problem is usually broken down into multiple subproblems that are interesting in their own right. Example subproblems include contaminant removal (filtering out sequencing reads that belong to contaminating bacteria rather than the target genome), error correction (flagging or fixing the errors introduced by the sequencing technology), overlap computation (constructing the overlap graph used for assembly), and construction of the de Bruijn graph (55) to represent the sequenced fragments. Transcript assembly is the related task of reconstructing the sequence of the transcripts from RNA-seq reads.

2.2.2. Variant calling. The aim of variant calling is to identify the genomic differences between a reference sequence and a sequenced individual, based on a collection of sequence fragments. Variant calling often requires a lower depth of sequencing than de novo genome assembly.

2.2.3. Read mapping. This is a first step in most variant calling pipelines. We are given a collection of fragments $\{t_1, \ldots, t_m\}$ and a reference string T, and our goal is to identify the best matching (typically under the edit distance or some appropriately defined alignment score) substring of T for each of the t_i. This problem arises when sequencing a new individual of a species for which a reference genome (T) is already known. Because of algorithmic decisions optimized to particular types of sequencing data, most sequence mappers only tackle one data type, although some tools are well suited to multiple tasks.

2.2.4. Sequence alignment. The goal of the sequence alignment problem is to find similar sequences, generally to determine if they are related in some way. A system such as BLAST (basic local alignment search tool) (56) or its compressive version CaBLAST (compressively accelerated BLAST) (25, 26) solves the problem: Given a set of sequences $\{t_1, \ldots, t_m\}$ and a query q, find the t_i and their substrings that are significantly similar to a substring of q. The software additionally reports the list of differences between the similar substrings of the t_i and of q. Variations require the entirety of q to be matched. When there is only one sequence in the set (i.e., $m = 1$) and t_1 and q both represent entire genomes, this is the genome-to-genome alignment problem.

2.2.5. Experiment search. Let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be a set of experiments, where each R_i is a set of short read fragments. The experiment search problem aims to find the subset of experiments from $\mathcal{R}$ for which it is likely that a query sequence q was among the sequences sampled. This problem has become more important recently, as larger collections of unassembled short read experiments have been deposited in public databases.

2.2.6. Metagenomic abundance estimation. In metagenomics, a large number of species from a common environment are sequenced together. The goal is to identify from this mixture which species or genes are present in what relative quantities in the environment.

3. COMPRESSED STRING INDEXES

The fundamental string processing problem in genomics is exact string search: Given strings T and P, with $|T| > |P|$, find the locations in T that match P exactly. This problem forms the basis of many approaches for sequence comparison and sequence mapping. To achieve speed, one typically indexes T by creating some data structure that allows various queries P to be processed quickly.

The suffix tree (57, 58), suffix array (59), enhanced suffix array (60), compressed suffix array (61), and FM-index (12) are all examples of full-text indices supporting exact string search. Although there are considerable differences in the manner in which these data structures work, they are all driven by the same underlying concept of allowing substring search by organizing all of the suffixes of the text. Since every substring of the text is a prefix of some suffix of the text, organizing and indexing the suffixes is sufficient to allow efficient (in the case of most of these structures, asymptotically optimal) pattern search.

This line of work also illustrates the need for sublinear algorithms: Earlier work on suffix trees uses $O(|T| + |P|)$ time for queries, but for modern data sizes this is too large. Suffix arrays, compressed suffix arrays, and the FM-index represent a line of work to reduce the space usage to the information-theoretic lower bound.

In the following, as is customary, when indexing n elements, we assume that manipulating pointers (memory words) of $\lg(n)$ bits takes constant time and constant space. In that sense, the suffix tree and suffix array are linear data structures, as they require $O(n)$ memory words for a text of n characters.

3.1. Suffix Trees and Suffix Arrays

The discovery of suffix trees is considered one of the defining discoveries in string algorithms. A suffix tree stores a text T using $O(|T|)$ space, it is constructed using $O(|T|)$ time ["The algorithm of the year," according to Knuth (58)], and it can answer a myriad (62) of text search queries in optimal asymptotic time. Once built, a suffix tree can determine if T contains P in time $O(|P|)$. If a pattern occurs k times in the text, then the suffix tree can enumerate all occurrences in $O(|P| + k)$ time, which is asymptotically optimal considering that the input must be read and the output must be written. Combined with the linear-time construction of the suffix tree, this proves that linear-time exact matching, $O(|T| + |P|)$, is possible.

While the suffix tree is still a conceptually important tool when thinking about string problems, it is rarely used in practice. The constant hidden in $O(|T|)$ is often too large in practice [around 20 bytes per node in the tree are typically required during construction for DNA alphabets (60)]. In contrast, the suffix array (59) can be constructed and stored in linear space, but with a more practical constant factor. However, it provides for asymptotically slower search than the suffix tree—pattern search is $O(|P| \log |T|)$, or $O(|P| + \log |T|)$ if one makes use of external longest common prefix (LCP) arrays. However, as demonstrated by Abouelhoda et al. (60), the enhanced suffix array, which consists of the suffix array augmented with external tables representing various components of the suffix tree topology, can answer all of the same queries as the suffix tree with

text *T*:	MISSISSIPPI$	
	IPPI$MISSISS	5
	ISSIPPI$MISS	8
	ISSISSIPPI$M	11
	I$MISSISSIPP	2
	MISSISSIPPI$	0
	PI$MISSISSIP	3
	PPI$MISSISSI	4
	SIPPI$MISSIS	6
	SISSIPPI$MIS	9
	SSIPPI$MISSI	7
	SSISSIPPI$MI	10
	$MISSISSIPPI	1
BWT(*T*):	SSMP$PISSIII	

Figure 1

Array A of Equation 1 for the classical example, "MISSISSIPPI." The suffix array is the right column of indices in blue. The Burrows–Wheeler transform (BWT) is the sequence of the red letters in the last column of the sorted rotations of the text.

the same asymptotic complexity. In such cases, the enhanced suffix array is preferred to the suffix tree, as it can be constructed and stored in less space and tends to provide memory access patterns that are practically more cache friendly.

The suffix array of T can be defined as follows. We assume that T ends with a special character "\$" not present elsewhere. Define the right rotation operation as $\mathcal{R}(T) = T[n]T[1 \ldots n-1]$, where n is the length of T. Construct the array (see **Figure 1**)

$$A = \left[\left(\mathcal{R}^i(T), i\right)\right]_{0 \le i \le n-1} \qquad 1.$$

consisting of all the pairs with right rotations of T and the amount of the rotation. Sort the array A using the lexicographic order of the first element of the pairs, the rotations of T. The suffix array SA(T) (59) is the list of the second element of the pairs in A, i.e., the list indices of the suffixes of T in lexicographic order. That is, if $i < j$, the suffix $T[\mathrm{SA}(T)[i] \ldots n-1]$ compares less lexicographically than the suffix $T[\mathrm{SA}(T)[j] \ldots n-1]$. The size of SA($T$) is $O(n)$ words. Searching for a substring in T using only the suffix array can be done in $O(|P| \log(|T|))$ time using a binary search on SA(T). It can be improved to search in time $O(|P| + \log(|T|))$ by the use of an additional data structure (LCP arrays).

3.2. The FM-Index and the Burrows–Wheeler Transform

Even suffix arrays are often too large to be practical for genomic scales, and they do not grow sublinearly. The FM-index (63) was called opportunistic by its creators, as it takes advantage of the redundancy of the input data to compress the data to store it, while retaining the ability to query the data structure with similar time guarantees as the uncompressed equivalent data structure. That is, the space requirement is $O(nH_k(T)) + o(n)$, where $H_k(T)$ is the kth-order entropy of the text T (for any k). The kth entropy is a measure of the information-theoretical minimum amount of space needed to encode each base of the sequence. In that sense, and up to a lower-order term, the space efficiency of the FM-index, as well as the compressed suffix array (61), is optimal.

Even though it is called "the" FM-index, it is rather a family of implementations with slightly different space and time guarantees, all starting from the Burrows–Wheeler transform (BWT) (64; see below) and space requirement bounded by the entropy of the text (whether H_0 or H_k for some range of k). See Reference 65 for a review of possible implementations. The efficiency of the FM-index and the compressed suffix array relies in part on the BWT.

Using the same array A defined in Equation 1, BWT(T) is the string of the last characters of the strings in A in the order they appear in A (see **Figure 1**). The size of BWT(T) is $O(|T|)$. The BWT(T) has many interesting properties. In particular, the transform is reversible—it is possible to recreate T from only BWT(T). Counting the number of occurrences of the pattern P in T can be performed in $O(|P|)$ time using only the BWT(T) and some small auxiliary information. The BWT has a few advantages compared to the suffix array: The BWT or associated information is usually smaller than the suffix array, and exact matches can be found without keeping the original text T. However, the BWT cannot determine locations of matches without, for example, a sampled suffix array (12, 65).

The string BWT(T) is amenable to further compression with standard compression techniques. The original FM-index implementation (63) uses a sequence of move-to-front coder (66), run-length coding, and variable-length prefix coding to compress BWT(T) to obtain a string of size $O(nH_k(T))$. In addition, the FM-index saves extra information, using only $o(n)$ space, so that for each query, it only needs to decode a small length of the compressed BWT. In some sense, the BWT encodes the same information as the suffix array, but in a pointerless manner (67) that can be more easily compressed while preserving the ability to recover the original string and run queries against it.

Transformations similar to BWT exist for objects besides strings. For example, the XBW transform (67) encodes trees, and encodings exist for directed acyclic graphs (68), planar graphs (69), and subgraphs of the de Bruijn graph (70).

3.3. Applications

The compressed text indices were originally developed for the problem of pattern searching in a text. The problem of sequence alignment is a more complicated version of pattern searching, and therefore these indices are naturally used in this context. Other applications such as genome assembly also make good use of these data structures.

3.3.1. Sequence alignment and read mapping.

Sequence alignment and read mapping are the most natural problems in which to apply compressed string indexes. The most common heuristic in sequence alignment is seed-and-extend, where first the aligner finds stretches of exact matches between the text and the queries, called seeds, then nearby seeds are clustered together, and the alignment is extended between the seeds. Many aligners follow this heuristic, with many variations on which seeds to use, how to cluster them, and how to extend the matches. Variant calling and transcript assembly rely on these aligners as the first step of their work.

For example, the `MUMmer3` (13) package, which is mostly used for aligning genome-to-genome, uses a suffix tree to find the seeds. Groups of closely spaced seeds are extended with dynamic programming. The newer and backward-compatible `MUMmer4` (14) uses a suffix array to find the seeds.

Many read mappers use ideas related to the FM-index, although in practice many of them only take advantage of linear space variants (rather than sublinear variants). Even this provides a smaller constant factor overhead than suffix arrays. To align short reads to genomes, Bowtie (15) uses an FM-index to minimize the memory footprint and speed up ungapped search. It combines the FM-index with a heuristic to permit a small number of mismatches. Bowtie 2 (16) includes both an FM-index and a dynamic programming element to handle longer inexact alignments without compromising speed. Other popular short read aligners that use the BWT and FM-index include

SOAP2 (21) and BWA (17). BWA-MEM (18) and BLASR (19) also employ the FM-index for aligning long reads.

Although the FM-index is relatively small and theoretically fast for exact searches, it suffers from poor cache locality: Every extension by a base looks at a different—and possibly distant—location of the BWT, typically incurring an expensive cache miss. To speed up search, the splice-aware aligner HISAT (20) uses a hierarchy of FM-indices. One large FM-index indexes the entire genome, while many small FM-indices (approximately 48,000 for the human genome) contain a small part of the genome. Each small FM-index is small enough to fit in the cache of a modern CPU. HISAT aligns small chunks of a read using the large (and slow) FM-index to find candidate matching locations. It then selects the appropriate small FM-indices to quickly extend the matches.

The "compressive genomics" framework (25, 26) exploits redundancies in the biological data to reduce the space required to store and the work required to analyze the data. In this framework, the time to perform a similarity search is bounded by the entropy and the fractal dimension of the high-dimensional data.

The CaBLAST suite of tools (25, 26) is functionally equivalent to the sequence aligner BLAST tools (71) but is faster and more memory efficient. As many sequences stored in a database are similar, CaBLAST splits the data into two databases. The coarse database contains the unique elements from the original database, while the remaining redundant elements are stored as pointers to a unique element and a list of differences.

The CORA software (27) is an accelerator for read alignment programs, such as BWA (17, 18) and Bowtie 2 (16). Instead of only using the redundancy in the database, CORA also uses the redundancy in the input query data. For example, in the problem of aligning sequencing reads to a genome, many of the input reads are very similar, as they are sequenced from the sequence part of the genome. CORA extracts common features of the input sequences, uses an external aligner like BWA or Bowtie 2 to align these features, and deduces the alignment of all the reads.

3.3.2. Genome assembly.

SGA (string graph assembler) (22, 72), an assembler for short reads, uses an FM-index at every step of the assembly pipeline. The first FM-index is constructed from the raw sequencing reads. The reads are partitioned in multiple groups and the FM-indexes are computed in parallel on each group to speed up construction. The FM-indexes are then merged (73). Directly using this FM-index, the SGA error corrector replaces a base in a read that is part of a low-coverage k-mer if replacing that base unambiguously improves the coverage of that k-mer. Then, using another FM-index, the contained reads are filtered out and the reads are merged to generate the string graph (74).

The DBGFM data structure (23) represents a de Bruijn graph used in genome assembly as an FM-index. This data structure was tested with the ABySS genome assembler (75).

4. SKETCHING AND LOCALITY-SENSITIVE HASHING

4.1. Locality-Sensitive Hashing

LSH was first introduced in the context of the nearest neighbor problem (76). In this problem, the goal is to efficiently search a set S of points in a high-dimensional metric space. A query consists of a point p in the same metric space, and the algorithm must return the point from S that is closest to the query point p. The approximate version of this problem relaxes this requirement and allows the algorithm to return a point not too far from the closest point. When the dimension of the metric space is high, this search is very expensive. LSH methods are probabilistic methods that quickly solve the approximate nearest neighbor problem.

It is possible to frame many bioinformatics problems as approximate nearest neighbor problems. For example, given a set of sequencing reads, it is beneficial to remove the reads coming from common bacterial contaminants. Aligning all the sequencing reads against the sequences of all the known contaminants is prohibitively costly. The contigs of the contaminant bacterial genomes can be seen as points in a high-dimensional space, and the approximate nearest neighbor algorithm finds the reads that are close to contaminant reads (40, 41). More precise alignment algorithms are then used on a much reduced set of reads to weed out the contaminating reads.

4.1.1. Definitions.

Given the universe of interest $\mathcal{U}$ (e.g., sequences, k-mers), a similarity measure S on $\mathcal{U}$ is a function with values in [0, 1], where higher values denote more similar elements. The Jaccard index is an example of a similarity measure. It measures how similar two sets are and is defined by $J(A, B) = |A \cap B|/|A \cup B|$. A metric (or semimetric) can define a similarity as well. For example, the Hamming distance H between two equal-length strings is the number of positions that differ between the two strings. For strings of length n, this defines the similarity $S_H(A, B) = 1 - H(A, B)/n$.

A family of hash functions is locality sensitive if it approximates a particular similarity, i.e., the hash functions are more likely to associate the same value to elements that are similar. Precisely, given a similarity measure S on the universe $\mathcal{U}$, a set $\mathcal{H}$ of hash functions defined on $\mathcal{U}$ is locality sensitive for S if

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = S(x, y), \qquad 2.$$

where the probability is taken over the distribution of hash functions. Then S is said to have or admit an LSH.

A different and more general definition is the gapped LSH (76). A distribution on $\mathcal{H}$ is (s_1, s_2, p_1, p_2)-sensitive if $S(x, y) \geq s_1$ implies that $\Pr[h(x) = h(y)] \geq p_1$ and $S(x, y) \leq s_2$ implies that $\Pr[h(x) = h(y)] \leq p_2$, where the probability is computed over the choice of hash function in $\mathcal{H}$. The interesting case is when we have $s_1 > s_2$ and $p_1 > p_2$. That is, elements that are very similar [$S(x, y) \geq s_1$] have a probability of at least p_1 of getting the same value, while elements that are not similar have a probability less than p_2 of getting the same value. The first definition of LSH is equivalent to a distribution that is (s_1, s_2, s_1, s_2)-sensitive for all s_1 and s_2. While a particular similarity measure may have multiple gapped LSHs, not all similarity measures have an LSH.

The Jaccard index admits an LSH. Consider the following functions: π is a permutation of the elements of $\mathcal{U}$—it is a function that associates a different integer in $[1, |\mathcal{U}|]$ to each element of $\mathcal{U}$—and $h_\pi(A) = \min_{x \in A} \pi(x)$ is the smallest value of an element in the set A according to the permutation π. The family of functions $\mathcal{H}_\pi = \{h_\pi \mid \pi \text{ a permutation of } \mathcal{U}\}$ is an LSH for the Jaccard index (76).

The Hamming similarity also has an LSH. Consider the family of projection functions P_i that select one character out of a string: $P_i(s_1 s_2 \ldots s_n) = s_i$. The set of all the projections $\mathcal{H}_H = \{P_i \mid i \in [1, n]\}$ is an LSH for the Hamming similarity H.

This projection-based LSH can be extended to projections that select k-mers instead of bases: $P_{k,i}(s_1 s_2 \ldots s_n) = s_i \ldots s_{i+k-1}$. Then, the family $\mathcal{H}_{H,k} = \{P_{k,i} \mid i \in [1, n - k + 1]\}$ is

$$\left(s_1, s_2, 1 - \frac{k(1 - s_1)}{1 - (k-1)/n}, \frac{s_2 - (k-1)/n}{1 - (k-1)/n}\right)\text{-sensitive}$$

for the Hamming similarity whenever we have $k(1 - s_1) \leq 1 - s_2$. A review of LSH functions can be found in Reference 77.

4.1.2. Sketches.

Using only one hash function gives a high-variance estimator of the measure. Instead, we typically create a sketch: For an element x, a sketch for the LSH is a vector $\langle h_1(x), \ldots, h_r(x) \rangle$ of hash values, where the hash functions are selected from $\mathcal{H}$ according to the LSH probability distribution. To compare two elements and get an estimation of their similarity, it suffices to look at the proportion of equal entries in their sketches as an estimate of the probability in Equation 2.

The LSH sketch formed in this way for the Jaccard index is called a MinHash sketch. In practice, finding r independent hash functions for the MinHash sketch is not easy. Instead, one can select a single hash function, and the sketch of set A consists of the r smallest values of $\{h(x) \mid x \in A\}$ or the r smallest values of the hash function in each of the partitions of the space of hash values (78).

4.2. Estimating Set Sizes

It is useful to be able to estimate the size of a set quickly and using little memory, in particular in cases where the set is too large to be stored in memory. The following probabilistic algorithms give an approximation of the number of distinct elements in a set (or, in a streaming setting, seen so far) while storing minimal information about the set itself. Most such algorithms are refinements on the following simple observation. First, define a hash function, $h : \mathcal{U} \rightarrow [0, s - 1]$, that maps uniformly the universe of elements of interest into an interval, and keep track of either the smallest hash value m or the largest number n of leading zeroes in the binary representations of the hash values. Then, the number of distinct elements in the set can be estimated as $(s - 1)/m$ or 2^n.

Again, using only one hash function leaves a high variance in the size estimation. By using r independent hashes, the standard deviation is divided by $\sqrt{r}$. However, finding r independent hash functions is not always practical. Practical algorithms avoid using r hash functions and provide better analysis of their performance. Many algorithms have been proposed for this problem; below we explain only a few of them.

An algorithm gives a (ϵ, δ)-approximation of the true size n of a set if the value returned n' satisfies $\Pr[|n' - n| \leq \epsilon n] \geq 1 - \delta$, where the probability is taken over the random choices internal to the algorithm. That is, it returns, with high probability, a good approximation of the size of the set.

The first of Bar-Yossef et al.'s (79) three proposed algorithms keeps track of the k lowest values found, where $k = O(1/\epsilon^2)$. This algorithm is an (ϵ, δ)-approximation using $O(\log(|U|)\log(1/\delta)/\epsilon^2)$ bits. They also proposed improvements that use only $O(1/\epsilon^2 + \log(|U|))$ bits. This last algorithm has a memory usage that is asymptotically optimal (80, 81). The MinCount algorithm (82) uses the kth minimum and combines it with a sublinear function (e.g., a logarithm) to obtain better precision (ϵ parameter) for a given amount of memory.

The HyperLogLog algorithm (83) splits the input stream into 2^p substreams, with the first p bits removed from the hash value and used for identifying which substream an element goes into. The max of these individual substreams is then determined and the cardinality is estimated as $\alpha_m \cdot m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]})$, where α_m equals $[m \int_0^\infty (\log_2(\frac{2+u}{1+u}))^m \, \mathrm{d}u]^{-1}$, $m = 2^p$ is the number of substreams, and $M[j]$ is the largest number of leading zeroes observed in substream j. The actual implementations of this method have undergone additional improvements such as the HyperLogLog++ algorithm (84), which adds a bias correction model and a heuristic for small cardinalities by implementing an alternative linear counting algorithm when HyperLogLog underperforms, both of which result in improved overall accuracy.

4.3. Estimating Distributions

The Count-Min Sketch algorithm (85) answers this query: How many times does element x occur in multiset S? It is an (ϵ, δ)-approximation algorithm that returns the number of occurrences of an element $x \in S$ with relative error less than ϵ with probability at least $1 - \delta$.

Count-Min Sketch selects $r = \lceil \lg(1/\delta) \rceil$ independent hash functions and stores a two-dimensional array with r rows and $w = O(1/\epsilon)$ columns, with each cell initially set to 0. Adding an element to the data structure is performed by adding 1 to one cell in each of the r rows. The index of the cell is given by the value of the hash functions for that element. Because of the one-sided error a hash collision can introduce, each cell is an overestimate of the number of occurrences of the elements hashing to that cell. The most accurate value for the element's frequency can be found by taking the minimum value in the cells identified by each of the r hash locations. This heuristic of taking the minimum value is similar to the recurring minimum heuristic in Spectral Bloom Filters (SBFs) (Section 5.2), and it is built on the idea that a minimum value has fewer hash collisions and is therefore more likely to be correct.

The ntCard algorithm (35) computes global frequency statistics. It answers the query: How many elements in S occur n times? (That is, it computes a frequency histogram.) ntCard uses only one hash function that gives p bits for each element in the input stream, using the ntHash algorithm. The input is sampled at some rate $1/2^q$ by keeping from the input stream only k-mers where the q largest bits in their hash values are 0. Each sampled element is identified with the r lowest bits in its hash value where $r \leq p - q$. ntCard stores an array of 2^r counters, all initialized to 0. If an element is sampled, the counter given by its hash value is incremented.

The ith counter in the array contains an overestimate of the number of occurrences of all elements hashing to i. Ideally, if r is very large (so large that 2^r is greater than the number of elements in the universe), then the ith counter would contain an exact value. As shown by Mohamadi et al. (35), by using independence properties between the various counters, the fraction f_i of elements occurring i times can be estimated by $\hat{f}_i = \frac{-p_i}{p_0 \lg p_0} - \frac{1}{i}\sum_{j=1}^{i-1}(j p_{i-j} \hat{f}_j)/p_0$, where p_i is counter i divided by the sum of all the 2^r counters.

More generally, given a set S of k-mers, let f_i be the number of k-mers occurring i times in S. The kth frequency moment is defined as $F_k = \sum_i i^k f_i$. F_0 is the number of distinct k-mers in S and F_1 equals $|S|$. The problem of approximating the moments F_k with a single-pass algorithm and with good accuracy has been heavily studied (82–87).

4.4. Applications

Dimensionality reduction methods can be applied to a large variety of problems in computational biology, as most data are high dimensional. The trade-off between loss of some amount of accuracy and gains in computation speed is beneficial in many applications.

4.4.1. Estimating *k*-mer distributions. In applications such as genome assembly (88) and sequence search (30), an estimation of the distribution of the k-mers, for varying values of k, is desirable for setting many parameters of the genome assembler.

LSH techniques are often used for this. For example, khmer (34) uses a Count-Min Sketch data structure to get an approximate number of occurrences for each k-mer. KmerStream (89) quickly estimates the number of distinct k-mers in a data set as well as the number of singleton k-mers, i.e., the k-mers occurring only once, which are likely to be due to sequencing errors. It combines subsampling the input k-mers and the size estimation algorithm described in Section 4.2 to obtain an accurate estimate. The ntCard (35) program uses the ntHash algorithm (which was developed for this purpose) to estimate the entire distribution of k-mer frequencies.

4.4.2. Read alignment. Read alignment using the seed-and-extend strategy involves two steps: First, find approximate candidate locations for alignment, and then refine the alignment if possible. LSH is very appropriate to implement the first step of this strategy.

The LSH-ALL-PAIRS (36) aligner does an all-versus-all alignment of a set C of sequences. To avoid the quadratic complexity of comparing every pair, it filters the likely candidate pairs using LSH for the Hamming distance. For every k-mer m in every sequence of C, tuples of the form $\langle h(m), \pi(m) \rangle$ are added to a list, where $h(m)$ is the hash value and $\pi(m)$ stores the information about the position of the k-mer in C. Then the list is sorted based on the hash values and reads with identical hash values are marked to be aligned more precisely. This procedure is repeated several times to avoid false negatives.

Although MHAP (37) is an overlapper and not a general read aligner, the overlap problem is really a special case of the all-versus-all alignment problem. MHAP also filters reads with a likely overlap using LSH for the Jaccard index. MHAP splits a read into k-mers, computes a hash value (giving a fingerprint), and keeps the smallest r fingerprints. It creates r hash tables such that the ith hash table maps a fingerprint to a list of reads where that fingerprint was the ith smallest. Finally, the reads that co-occur in the same lists in a large enough percentage of the r hash tables are compared more precisely. The ideas of MHAP have been applied specifically to read alignment in MashMap (38).

4.4.3. Metagenomics abundance estimation. The problem of metagenomics abundance estimation is to determine the number of species in a sample from a mixture of sequences. One approach is to group together similar sequences, which presumably come from related species.

LSH-Div (39) uses an LSH approximating the Hamming distance to do this. Effectively, all pairs of sequences are compared, and sequences that are significantly different are put in different bins. The procedure is repeated multiple times with different random hash functions to minimize the false negative rate. At each iteration, bins that are found to be similar according to the new hash function are merged.

The program Mash (40) and the utility sourmash (41) proceed in a similar fashion, but using MinHash sketches. Sketches are created for the sequences, which creates signatures thousands of times smaller than the original sequences (just a few kilobytes per sketch, regardless of the original sequence size). For the clustering itself, every pair of sequences is compared using the Jaccard index LSH approximation. Even though this pairwise comparison is quadratic in the number of sequences, given that each comparison is very fast and requires little input/output, it is possible to cluster thousands of sequences in this way.

5. APPROXIMATE MEMBERSHIP QUERIES

The compressed indices of Section 3 give the same answer as the uncompressed versions of the same queries. If the application can tolerate some errors introduced by the data structure, then much more memory-frugal, and potentially faster, data structures exist.

AMQ structures comprise one class of data structures that have proven particularly useful for this. These data structures store a set S and support at least the following operations: `Insert`(S, x), which adds item x from a universe $\mathcal{U}$, and `Contains`(S, x), which determines whether the current S contains $x \in \mathcal{U}$.

AMQs are allowed to produce errors on the `Contains` operation. Typically, only one-sided error is allowed: If x is in S, `Contains`(S, x) must return `true`. If x is not in S, `Contains`(S, x) may incorrectly return `true` with some probability. For this data structure to be useful, the false positive rate, defined as the probability of `Contains` returning `true` for a random element that is not in the set, must be bounded. Some AMQs allow an additional `Delete` operation or are able to store multisets by storing an approximate count for each item.

AMQ data structures are typically applied in genomics by taking the set S that they encode to be the set of k-mers that are present in the strings of interest. A query string $s_1 s_2 \ldots s_n$ can be tested for membership in the indexed strings by $\texttt{Contains}(S, s[i \ldots i+k-1])$ for each i. If a sufficient number of these operations return `true`, then it is likely that a string similar to s is present.

The one-sided error of AMQ data structures often can be mitigated. If an application is estimating some properties of the sequences, such as the number of distinct k-mers or the frequency histogram of k-mers, then the space and speed benefit of an AMQ typically outweigh the small decrease in accuracy of the estimates. In many bioinformatics applications, numerous convergent pieces of information are needed to reach a particular conclusion. For example, if a read contains a small k-mer in common with some bacterium, this does not immediately mean that the read is a contaminant, as this can happen by chance. The additional errors created by the AMQ are then incorporated in the larger error model (covering sequencing errors, random similarities, etc.) used to distinguish real from spurious events. Other applications use an AMQ in a way that errors have no impact on the ultimate result by using the AMQ as a prefilter to avoid costly computation, but ultimately verifying each query done by the AMQ.

5.1. Bloom Filters

The Bloom filter (90) is a probabilistic data structure that stores an arbitrary set and supports the two AMQ operations `Insert` and `Contains`. A Bloom filter consists of an m-bit array B, as well as r hash functions $h_i : \mathcal{U} \to [0, m-1]$, for $i = 1, \ldots, r$. Here, r and m are tunable parameters that affect the space usage and the false positive rate.

$\texttt{Insert}(x)$ sets $B[h_i(x)]$ to 1 for each $1 \leq i \leq r$. $\texttt{Contains}(x)$ returns `true` only if $B[h_i(x)]$ equals 1 for each i. This introduces no false negatives but may introduce false positives if all of the $B[h_i(x)]$ bits have been set to 1 by `Insert`s not involving x. The variables r and m can be set to control how often this is expected to happen (91): For n inserted elements, the false positive probability is approximately $(1 - e^{-rn/m})^r$. If m is fixed by desired memory usage, and if we expect n distinct elements to be inserted, the number of hash functions that minimizes the false positive rate is $k = (m/n) \ln 2$.

5.2. Counting Bloom Filters and Spectral Bloom Filters

Counting Bloom filters (CBFs) extend the Bloom filter by encoding an approximate count value for each element in the set. This was first introduced in the application of web cache sharing (92), where four bits were used in place of each bit in B to encode counts of up to 15. `Insert` now increases the count value at each hash position by 1, up to the maximum possible value. A new function `Delete` decrements those same counts, except when the counter is at its maximum value (to avoid undercounting and false negatives). `Contains` follows the same procedure as the base Bloom filter, except it returns the smallest count value found at any hash positions.

SBFs (93) solve a similar problem as counting Bloom filters but for instances where the counts may be significantly larger than the values allowed in a CBF. Like a CBF, the data structure is an array where each hash position is a count starting at 0 rather than a single bit. Rather than increment all r hash positions, an SBF `Insert` operation only increments the position(s) with the minimum current value, a heuristic known as minimal increase. An SBF `Contains` operation returns the minimum value over the positions indicated by the hash values. SBFs implement an additional heuristic, called recurring minimum, which identifies items that have few hash collisions by the presence of a recurring minimum count among their different stored values. The heuristic assumes that these items have a smaller than average error rate. Items where the number of occurrences of their minimum value is small among their hash locations are at risk of being false positives. The recurring minimum heuristic encodes those items with only a single occurrence of their minimum value in a secondary SBF, thereby reducing the average error.

5.3. Cascading Bloom Filters

Cascading Bloom filters (94, 95) were developed primarily as a way of improving the accuracy of a Bloom filter by recursively storing the set of false positives present in the previous filter. Specifically, let K_0 be the set of k-mers that are to be stored, and let B_0 be a standard Bloom filter into which these k-mers have been inserted. A cascading Bloom filter stores an additional Bloom filter B_1 into which some critical elements of interest in the set $\{x \mid \texttt{Contains}(B_0, x) = \texttt{true} \text{ and } x \notin K_0\}$ have been inserted. B_1 therefore stores the false positives of B_0 that are important to the application. The `Contains` operation of a cascading Bloom filter returns `true` if $\texttt{Contains}(B_0, x)$ returns `true` and $\texttt{Contains}(B_1, x)$ returns `false`.

Unfortunately, B_1 also generates false positives (which would become false negatives of the cascading Bloom filter if left unchecked), so this process continues recursively until there are no more errors on critical elements: B_2 is created to contain the elements in B_1 that are falsely identified as false positives. B_i for $i > 2$ are created using the same logic, terminating in an empty Bloom filter. $\texttt{Contains}(x)$ returns `true` if and only if there exists an odd j such that $\texttt{Contains}(B_i)$ returns `true` for $i < j$ and $\texttt{Contains}(B_j)$ returns `false`. As each of the successive filters B_i is storing a smaller set, they can decrease in size. As this recursive construction step identifies false positives after the construction of the entire filter, unlike the Bloom filter, the cascading Bloom filter is not suitable for one-pass streaming algorithms.

5.4. Hierarchical Bloom Filters

When dealing with multiple genomes or experiments, we have multiple sets $S_1, \ldots, S_m$ that we want to store. The desired $\texttt{SubsetThatContains}(\{S_1, \ldots, S_m\}, x)$ operation returns the set $\{i \mid \texttt{Contains}(S_i, x) = \texttt{true}\}$. If the sets S_i have some overlap in the k-mers they contain, a hierarchical Bloom filter approach may allow `SubsetThatContains` queries to be processed more efficiently.

Hierarchical Bloom filters consist of a rooted tree $\mathcal{T}$ with leaves corresponding to each set S_i. A standard Bloom filter B_i representing S_i is stored at leaf i. An internal node u is associated with a Bloom filter B_u storing the union of the sets stored at the children of u. B_u can be constructed by taking the bitwise `OR` of all of u's children. By construction, if $\texttt{Contains}(B_u, x)$ is `false`, then x is not present in any leaf in the subtree rooted at u. The $\texttt{SubsetThatContains}(x)$ operation begins at the root of $\mathcal{T}$ and traverses the tree, exploring only subtrees rooted at nodes u where $\texttt{Contains}(B_u, x)$ is `true`.

If the leaves in the tree are clustered by similarity, this index allows many sets that do not contain the query to be eliminated at the same time, since the absence of a given query can be identified in many leaves simultaneously by querying their lowest common ancestor once. Several different schemes have been proposed to select the topology of the tree $\mathcal{T}$, including B+ trees (96, 97) and greedily constructed binary trees (30). In addition, more efficient representations such as the Split Sequence Bloom Tree (SSBT) (31) and All-Some Sequence Bloom Tree (SBT-ALSO) (98) further reduce the storage cost and query times by eliminating redundancy in the Bloom filters stored along any path from the root to a leaf.

5.5. Quotient Filters and Counting Quotient Filters

The quotient filter (QF) (99) is another data structure for efficiently storing a set of elements. Let E be a set of elements, $h(\cdot)$ a hash function, and $h(E)$ the set $\{h(e) \mid e \in E\}$. A QF is a lossless encoding of $h(E)$. It is based on the principle of quotienting, originally introduced by Knuth (100), which uses the location of hash entries to reduce the storage of hashed items. Let $h(\cdot)$ be a p-bit hash function so that, for all e, $h(e)$ is p bits long. We define the q most significant bits of $h(e)$ to be the quotient of $h(e)$, denoted q_e, and the $r = p - q$ remaining bits to be the remainder of $h(e)$, denoted r_e. To store the elements, the QF uses an array of 2^q slots, each of size r bits. Each slot is associated with additional metadata bits that assist in resolving collisions and in querying the filter. Like the Bloom filter, the QF supports insertion and containment queries. Additionally, it supports removal of keys and enumeration of the set it encodes [i.e., enumeration of $h(E)$], which the Bloom filter does not. These extra capabilities make the QF more flexible than the Bloom filter.

The counting QF (CQF) (101) is a data structure for efficiently storing a multiset of elements. It works in a similar manner to the QF, but it allows one to associate with every $h(e)$ some multiplicity $m[h(e)]$. Rather than use an auxiliary array to maintain counts, the CQF directly uses empty slots that are normally used to store the remainders of keys (i.e., r_e). The counter for an element is stored adjacent to the corresponding remainder in the filter and is encoded in base $2^r - 1$. The CQF itself is based on a modification of the QF called the rank-and-select QF (RSQF) (101), which uses rank-and-select operations on the metadata bits to improve the space and time efficiency of the QF. This data structure requires $(2.125 + \log_2 \frac{1}{\delta})/\alpha$ bits per element, where δ is the false positive rate and α is the load factor of the filter. The RSQF (and CQF) operates up to load factors of 0.95, making this data structure more memory efficient than the Bloom filter when δ is less than $1/64$.

5.6. Applications

Many conventional methods of analysis in computational biology rely on heuristics for efficient but inexact results. Approximate membership testing is particularly relevant in these contexts, as the additional loss of precision can often be amortized at scale or addressed through postprocessing.

5.6.1. Counting k-mers.

Counting, or approximately counting, the number of occurrences of every k-mer in a set of sequences is a common step in many pipelines. The frequency of the k-mers in input sequences can be used to find or quantify repeat regions and sequencing errors and estimate the size of the underlying genome. When k is moderately large, counting the number of occurrences of each k-mer in large input sets is resource intensive, and many computational techniques have been devised for that problem. AMQ data structures have been one of those techniques.

In many applications, there is a large number of k-mers with small counts, often due to sequencing errors. These single-count k-mers do not require any additional count information, saving considerable space. Many k-mer counting tools handle these low-count k-mers using an AMQ data structure to separate single-occurrence k-mers from multi-occurrence k-mers whose counts must be stored. Software like Jellyfish (28) and BFCounter (29) uses a standard Bloom filter to that effect.

Although these approaches can introduce false positive k-mers with count 1, many downstream analyses discard single-count k-mers as erroneous, assuming that they represent sequencing errors. So these probabilistic methods offer significant efficiency gains at minimal cost.

5.6.2. Experiment discovery. We say that a set R of unassembled reads contains a query sequence q if there is some subset of the reads in R that can approximately cover q. Given a large set of experiments $\mathcal{R} = \{R_1, \dots, R_m\}$, we often want to identify which R_i contain q. For example the R_i could be RNA-seq experiments and q could be a novel transcript. The R_i that contain q are experiments that likely express this new transcript. Hierarchical Bloom filters can be used to solve this problem.

The first approach to this problem was the Sequence Bloom Tree (SBT) (30). An SBT encodes each experiment R_i as a compressed Bloom filter of the set of k-mers contained in R_i. These compressed Bloom filters are stored at leaves of a binary tree and internal filters are created, as described in Section 5.4. SBTs use the RRR (Raman–Raman–Rao) (102) compression scheme on the Bloom filters, which allows for individual bits to be queried in the compressed filters without having to decompress each filter. During the search for q, subtrees are pruned if an insufficient number of the k-mers of q are present at the root filter of the subtree. This is similar to the approach of Bloofi (96, 97) but optimized for sequencing data. The SBT was further improved with both SSBT (31) and SBT-ALSO (98). These latter techniques store two Bloom filters at each node u: One Bloom filter contains the k-mers that are present in all leaves under u, while the other filter contains those k-mers that are present in some but not all experiments. These two filters avoid needing to explore subtrees in which all leaves contain the query q, a significant advantage for commonly occurring queries.

Alternative approaches to AMQ-enabled experiment search are given by SeqOthello (103) and Mantis (33). These indexing strategies map each k-mer to a set of experiments containing it. The query times of these approaches scale with the number of distinct k-mers in a query, rather than the number of experiments in a data set. Mantis uses CQFs (Section 5.5) to further improve the accuracy of its index on the single-k-mer level.

5.6.3. Genome assembly with de Bruijn graphs. The set of k-mers present in an input sample implicitly encodes a graph, called the de Bruijn graph, which has proven to be an incredibly important combinatorial structure in genomics (55, 104, 105). Specifically, let $S = \{m_1, \dots, m_n\}$ be a set of k-mers present in a sample. The edge-centric de Bruijn graph is then a graph in which each k-mer induces an edge $m = (\ell(m), r(m))$, where $\ell(m)$ is the leftmost $(k-1)$-mer of m and $r(m)$ is the rightmost $(k-1)$-mer of k. When constructed from sequencing reads or a reference genome, the de Bruijn graph has the property that each input sequence can be spelled out via a path in the graph. These graphs have therefore formed the basis of many genome assembly approaches (55, 104). Sequence contexts of length larger than k are not necessarily retained and so there may be many walks in the graph that do not spell out any input sequence. Nonetheless, the de Bruijn graph contains considerable information about the input sequences, and it can be constructed and represented efficiently (often using the k-mer counting techniques described above). Nevertheless, an explicit representation of this graph is often large and unwieldy to work with.

This has given rise to probabilistic de Bruijn graphs (106) that exploit the correspondence between a k-mer set and the graph. Probabilistic de Bruijn graphs represent the graph using a set of Bloom filters to record the edges (k-mers) that are present in the graph. The outgoing edges of a $(k-1)$-mer can be found by querying the Bloom filter for every possible single-base extension. Cascading Bloom filters have also been used to exactly represent these graphs (94). A probabilistic representation of the weighted de Bruijn graph, relevant for transcriptome assembly, has been proposed (107). These approaches allow the de Bruijn graph to be stored and traversed much more efficiently.

6. MINIMIZERS

The minimizers scheme is another approach to create a small representation of a collection of sequences. It was first introduced to help compute read overlaps when Sanger sequencing (108) was most common. An equivalent method known as winnowing was introduced for document fingerprinting and plagiarism detection (109). Since these original applications, the minimizer idea has found a number of applications in speeding up sequence analysis. Like LSH, the minimizers scheme and its generalizations called local schemes are used to capture the likely similarity of sequences. LSH and minimizers schemes are sometimes used together.

6.1. Definition

The minimizers scheme has three parameters: k, the k-mer length; w, the length of the window; and π, an order (i.e., a permutation) of the k-mers. Given a sequence S, the minimizers scheme will select, in every overlapping window of w consecutive k-mers, the smallest k-mer according to π (choosing the leftmost k-mer in case of ties). Depending on the application, the result of the minimizers scheme is either the locations of the selected k-mers in S or the sequences of the selected k-mers (see **Figure 2**).

A local scheme is a generalization of the minimizers scheme. It also has three parameters: k, w, and a function f taking w k-mers as arguments and returning an index in $[1, w]$. For every window of w consecutive k-mers, the local scheme calls the function f with the w k-mers as arguments and selects the k-mer at the position returned by f. A minimizers scheme is therefore a local scheme where the function f_π returns the index of the leftmost, smallest k-mer according to some permutation π.

Local schemes (including minimizers) select k-mers such that the following two properties hold: (*a*) two consecutive selected k-mers are no more than w bases apart, and (*b*) sequences with an exact match at least $w + k - 1$ bases long will have a common selected k-mer. The first property is true because a k-mer is selected in every w-long window. It ensures that no part of a sequence is ignored. The second property is satisfied because the local scheme only looks at the sequence within a window to select a k-mer. Hence, when two sequences have an exact match that is at least as long as a window, the same k-mer is selected in that common subsequence.

Although the minimizers method is similar to LSH and has uses that overlap with LSH, it is not directly an LSH. As described above, the minimizers scheme does not define a similarity measure.

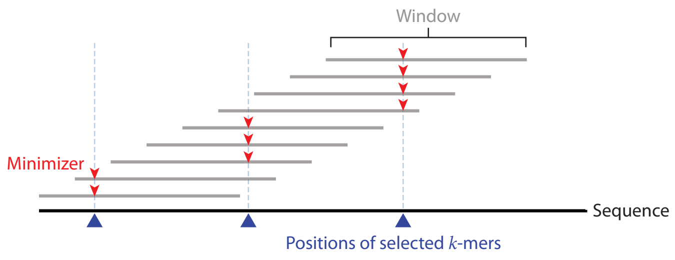

Figure 2

Given a sequence (*thick black line*), in each window (*gray lines*), a minimizer is selected (*red marks*). Consecutive windows are likely to select the same minimizers, and the selected positions constitute a sampling of the original sequence (*blue triangles*).

6.2. Parameters of Minimizers Schemes

The density of a minimizers scheme is the number of positions chosen over the length of the input sequence. It is computed as an expectation over an infinite length sequence with bases chosen independently at random. Depending on the application, low density implies reducing the size of the bins or the number of minimizers positions to keep in a hash, which in turn improves the runtime or memory usage.

The parameters w and k in the minimizers or local scheme are constrained by the application. A large value of k, and hence long k-mers, avoids spurious collisions, while a large value of w lowers the density, and the length of the windows $w + k - 1$ is the exact match requirement. The order π on the k-mers (or function f for a local scheme) is mostly unconstrained: Regardless of the choice of the order or function, the above properties a and b of minimizers are satisfied. Consequently, improving the density of minimizers schemes by choosing the best π is of great interest, as any improvement in density would benefit existing and future applications with little change to the core algorithm of the applications.

Because at least one k-mer is chosen in every w-long window, the density is necessarily at least $1/w$. A few orders on k-mers perform poorly, such as the lexicographic order on k-mers, and this was recognized very early on (110). A random permutation of the k-mer performs reasonably well in practice and is the usual choice. For a random permutation and for w not too large ($w \ll \sigma^k$, where σ is the size of the alphabet), the expected density is $2/(w + 1)$, which is almost twice the optimal density of $1/w$.

The theory behind minimizers and, even more so, local schemes is not yet fully understood. In particular, the problem of creating orderings or functions giving the lowest possible density is still open. It is possible (111, 112) to create orders with density lower than $2/(w + 1)$, but the construction is not always practical, as it requires storing a large set of k-mers. Orderings achieving the optimal density of $1/w$ do exist (113), but the construction is optimal only in the asymptotic case where w is fixed and k becomes very large. For the more common case where w is larger than k, minimizers schemes cannot achieve the optimal density (113). Local schemes, on the other hand, do not have this limitation and can potentially achieve optimal density. This is an active domain of research.

6.3. Applications

Minimizers were first introduced in 2003 for the computation of Sanger read overlaps. It took another ten years for the method to be used in other applications. Since 2013, many methods have used minimizers to speed up their operations.

6.3.1. Pairwise string overlaps. Pairwise string overlaps is the original genomics application for which minimizers were invented (108, 110): Given a set of sequencing reads, the first step of the overlap–layout–consensus method for genome assembly (114) is to find all the read pairs where the suffix of one read aligns with the prefix of the other. For n reads, aligning every pair of reads requires $O(n^2)$ calls to the aligner, which is very expensive computationally.

To speed up this approach, one first bins together the reads using minimizers such that two reads with a possible overlap are in at least one common bin, while reads with no overlap are likely not to share any bin. To do this, the overlapper puts each read s in bins corresponding to all the minimizer k-mers of s. The more precise and time-consuming alignment step needs to be run only between reads within each bin.

If two reads have an overlap of a minimum quality (minimum length with minimum identity), they must have an exact match of a reasonable length ℓ. The parameters k and w are chosen so that ℓ equals $w + k - 1$; hence, property *b* of the minimizers guarantees that two reads with a long enough overlap share at least one bin. If k is large enough, it is unlikely that two reads that do not have an exact match have the same minimizers, and this reduces the size of the bins and the number of unnecessary comparisons.

6.3.2. Read alignment. The aligners minimap (42) and minimap2 (43) are designed to align very long reads that have high error rates. These aligners use minimizers to anchor the alignment. First, all the minimizers of the reference sequence are computed and entered into a hash table, where the key is the sequence of the minimizer, and the value is a list of all the indices in the reference sequence where this k-mer is selected as the minimizer. In aligning a query sequence, the minimizers of the query are searched in the hash table for locations where the query might align.

6.3.3. Counting *k*-mers with super *k*-mers. Some fast k-mer counters also use minimizers and binning to parallelize their operation (44, 45, 115, 116), although in a very different way than for overlap computation. These applications take a set of sequences (e.g., sequencing reads, genomes) and count the number of occurrences of every k-mer present in the sequences. Here, the minimizers are m-mers, with m typically in the range [7, 12], and the k-mers that are to be counted have length k, which is typically larger than 14 and up to 70. The parameter w is defined by the relation $w + m - 1 = k$, so that a window of w consecutive m-mers represents one k-mer.

The k-mer counter application breaks the input sequences into so-called super k-mers. A super k-mer is a sequence of consecutive windows that select the same m-mer as a minimizer. Each super k-mer is binned according to the minimizer in it. Finally, the application tallies the number of occurrences of each k-mer in each bin.

Property *b* of the k-mers and the choice of w guarantee that, for every k-mer x, the super k-mers with a copy of x are binned together in one single bin. Because a k-mer cannot be present in two different bins, it is possible to perform the counting in each bin independently, and the problem becomes embarrassingly parallel.

6.3.4. De Bruijn graph creation. The two variants of the BCALM algorithm (23, 24) assemble the nonbranching parts of the assembly de Bruijn graph (known variously as maximal unitigs, unipaths, and k-unitigs) using an idea similar to super k-mers. The k-mers in the sequence reads are separated into bins based on their minimizers. Then the algorithm creates the maximal unitigs in each of the bins. Finally, the algorithm glues together paths from different bins by proceeding in a particular order to guarantee that all unitigs are created. The two versions of the algorithm vary on some important details, but both use the properties of minimizers to efficiently split the work of maximal unitigs creation and to efficiently find the bins with paths that need to be glued.

6.3.5. Sparse data structures. Sparse data structures can use minimizers to sample sequences and only store information at certain designated points, thereby reducing the amount of information to store. The two properties of minimizers (Section 6.1) ensure that two sequences that have a significant exact match will select the same minimizers. This guarantees that, down to some minimum resolution, the same information is stored in the sparse data structure as in the full data structure.

The SparseAssembler (46) for short read assembly uses a de Bruijn graph constructed only on the minimizers of the sequencing reads, with edges representing not just a shift by one base but also the sequence joining two consecutive minimizers. The remainder of the algorithm to simplify the graph and generate the contigs from the de Bruijn graph is otherwise comparable to other de Bruijn graph–based assemblers. Although in this case sequencing errors could cause some branches to be missed and therefore could introduce assembly errors, especially for large window sizes, in practice, window sizes of 10 to 15 show little to no degradation and significant memory savings.

The suffix array data structure (Section 3.1) is also amenable to downsampling with minimizers (47). Instead of keeping an index for every suffix in the text T, only the indices starting at minimizers are kept. In aligning a pattern, the minimizers are also extracted from the pattern and searched for in the suffix array.

The MashMap aligner (38) aligns long reads to a genome. It combines the minimizers method with MinHash sketches (Section 4.1.2) to further reduce the size of the index. The index created on the genome consists of the hash value and position computed for every minimizer found in the genome. In aligning a sequence q, the minimizers are also extracted from q, and at every location with a shared minimizer between q and the genome, the Jaccard index is estimated using the MinHash sketch method. This provides the seeds for a seed-and-extend alignment strategy.

7. CONCLUSION

Compact, sublinear data structures have been an essential component of genomics tool building. Research into how to employ existing structures and the design of novel techniques has propelled more efficient solutions to problems such as read mapping and genome assembly and has spurred the formulation of new problems such as experiment discovery. The key idea of all of these approaches is to create data structures that store the relevant aspects of the underlying sequences in as small a space as possible while still supporting the operations necessary for the biological application. Research continues in each of these areas, and it is likely that new approaches will be developed for existing and new sequence analysis problems.

DISCLOSURE STATEMENT

R.P. and C.K. are cofounders of Ocean Genomics, Inc.

ACKNOWLEDGMENTS

This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through grant GBMF4554 to Carl Kingsford, by the US National Science Foundation (BIO-1564917, CCF-1750472, and CNS-1763680), and by the US National Institutes of Health (R01HG007104, R01GM122935, and R01HG009937).

LITERATURE CITED

1. Stephens ZD, Lee SY, Faghri F, Campbell RH, Zhai C, et al. 2015. Big data: Astronomical or genomical? *PLOS Biol.* 13:e1002195
2. 1000 Genomes Proj. Consort. 2015. A global reference for human genetic variation. *Nature* 526:68–74
3. Sudmant PH, Rausch T, Gardner EJ, Handsaker RE, Abyzov A, et al. 2015. An integrated map of structural variation in 2,504 human genomes. *Nature* 526:75–81
4. Weinstein JN, Collisson EA, Mills GB, Shaw KRM, Ozenberger BA, et al. 2013. The Cancer Genome Atlas pan-cancer analysis project. *Nat. Genet.* 45:1113–20
5. Wang Z, Gerstein M, Snyder M. 2009. RNA-Seq: a revolutionary tool for transcriptomics. *Nat. Rev. Genet.* 10:57–63

6. Edgar R, Domrachev M, Lash AE. 2002. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic Acids Res.* 30:207–10
7. Leinonen R, Sugawara H, Shumway M. 2011. The sequence read archive. *Nucleic Acids Res.* 39:D19–21
8. Stoesser G, Baker W, van den Broek A, Camon E, Garcia-Pastor M, et al. 2002. The EMBL Nucleotide Sequence Database. *Nucleic Acids Res.* 30:21–26
9. Leinonen R, Akhtar R, Birney E, Bower L, Cerdeno-Tárraga A, et al. 2010. The European Nucleotide Archive. *Nucleic Acids Res.* 39:D28–31
10. Loh PR, Baym M, Berger B. 2012. Compressive genomics. *Nat. Biotechnol.* 30:627–30
11. Berger B, Daniels NM, Yu YW. 2016. Computational biology in the 21st century: scaling with compressive algorithms. *Commun. ACM* 59:72–80
12. Ferragina P, Manzini G. 2005. Indexing compressed text. *J. ACM* 52:552–81
13. Kurtz S, Phillippy A, Delcher AL, Smoot M, Shumway M, et al. 2004. Versatile and open software for comparing large genomes. *Genome Biol.* 5:R12
14. Marçais G, Delcher AL, Phillippy AM, Coston R, Salzberg SL, Zimin A. 2018. MUMmer4: a fast and versatile genome alignment system. *PLOS Comput. Biol.* 14:e1005944
15. Langmead B, Trapnell C, Pop M, Salzberg SL. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.* 10:R25
16. Langmead B, Salzberg SL. 2012. Fast gapped-read alignment with Bowtie 2. *Nat. Methods* 9:357–59
17. Li H, Durbin R. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 25:1754–60
18. Li H, Durbin R. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26:589–95
19. Chaisson MJ, Tesler G. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics* 13:238
20. Kim D, Langmead B, Salzberg SL. 2015. HISAT: a fast spliced aligner with low memory requirements. *Nat. Methods* 12:357–60
21. Li R, Yu C, Li Y, Lam TW, Yiu SM, et al. 2009. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics* 25:1966–67
22. Simpson JT, Durbin R. 2012. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.* 22:549–56
23. Chikhi R, Limasset A, Jackman S, Simpson JT, Medvedev P. 2015. On the representation of de Bruijn graphs. *J. Comput. Biol.* 22:336–52
24. Chikhi R, Limasset A, Medvedev P. 2016. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics* 32:i201–8
25. Yu YW, Daniels NM, Danko DC, Berger B. 2015. Entropy-scaling search of massive biological data. *Cell Syst.* 1:130–40
26. Daniels NM, Gallant A, Peng J, Cowen LJ, Baym M, Berger B. 2013. Compressive genomics for protein databases. *Bioinformatics* 29:i283–90
27. Yorukoglu D, Yu YW, Peng J, Berger B. 2016. Compressive mapping for next-generation sequencing. *Nat. Biotechnol.* 34:374–76
28. Marçais G, Kingsford C. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinformatics* 27:764–70
29. Melsted P, Pritchard JK. 2011. Efficient counting of *k*-mers in DNA sequences using a Bloom filter. *BMC Bioinform.* 12:333
30. Solomon B, Kingsford C. 2016. Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.* 34:300–2
31. Solomon B, Kingsford C. 2018. Improved search of large transcriptomic sequencing databases using split sequence Bloom trees. *J. Comput. Biol.* 25:755–65
32. Sun C, Harris RS, Chikhi R, Medvedev P. 2018. AllSome sequence bloom trees. *J. Comput. Biol.* 25:467–79
33. Pandey P, Almodaresi F, Bender MA, Ferdman M, Johnson R, Patro R. 2018. Mantis: a fast, small, and exact large-scale sequence-search index. *Cell Syst.* 7:201–7.e4

34. Zhang Q, Pell J, Canino-Koning R, Howe AC, Brown CT. 2014. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PLOS ONE* 9:e101271
35. Mohamadi H, Khan H, Birol I. 2017. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics* 33:1324–30
36. Buhler J. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17:419–28
37. Berlin K, Koren S, Chin CS, Drake JP, Landolin JM, Phillippy AM. 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.* 33:623–30
38. Jain C, Dilthey A, Koren S, Aluru S, Phillippy AM. 2017. A fast approximate algorithm for mapping long reads to large reference databases. In *Research in Computational Molecular Biology*, ed. SC Sahinalp, pp. 66–81. Cham, Switz.: Springer Int.
39. Rasheed Z, Rangwala H, Barbara D. 2012. LSH-Div: species diversity estimation using locality sensitive hashing. In *2012 IEEE International Conference on Bioinformatics and Biomedicine*, ed. J Gao, W Dubitzky, C Wu, M Liebman, R Alhaij, et al. New York: IEEE
40. Ondov BD, Treangen TJ, Melsted P, Mallonee AB, Bergman NH, et al. 2016. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biol.* 17:132
41. Brown CT, Irber L. 2016. sourmash: a library for MinHash sketching of DNA. *J. Open Source Software* 1:27
42. Li H. 2016. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* 32:2103–10
43. Li H, Birol I. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34:3094–100
44. Li Y, Yan X. 2015. MSPKmerCounter: a fast and memory efficient approach for k-mer counting. arXiv:1505.06550 [q-bio.GN]
45. Deorowicz S, Kokot M, Grabowski S, Debudaj-Grabysz A. 2015. KMC 2: fast and resource-frugal *k*-mer counting. *Bioinformatics* 31:1569–76
46. Ye C, Ma ZS, Cannon CH, Pop M, Yu DW. 2012. Exploiting sparseness in *de novo* genome assembly. *BMC Bioinform.* 13:S1
47. Grabowski S, Raniszewski M. 2015. Sampling the suffix array with minimizers. In *String Processing and Information Retrieval*, ed. C Iliopoulos, S Puglisi, E Yilmaz, pp. 287–98. Cham, Switz.: Springer Int.
48. Merriman B, Ion Torrent R&D Team, Rothberg JM. 2012. Progress in Ion Torrent semiconductor chip based sequencing. *Electrophoresis* 33:3397–417
49. Shendure J, Porreca GJ, Reppas NB, Lin X, McCutcheon JP, et al. 2005. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science* 309:1728–32
50. Bennett S. 2004. Solexa Ltd. *Pharmacogenomics* 5:433–38
51. Jain M, Olsen HE, Paten B, Akeson M. 2016. The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community. *Genome Biol.* 17:239
52. Quail MA, Smith M, Coupland P, Otto TD, Harris SR, et al. 2012. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genom.* 13:341
53. Clarke J, Wu HC, Jayasinghe L, Patel A, Reid S, Bayley H. 2009. Continuous base identification for single-molecule nanopore DNA sequencing. *Nat. Nanotechnol.* 4:265–70
54. Eid J, Fehr A, Gray J, Luong K, Lyle J, et al. 2009. Real-time DNA sequencing from single polymerase molecules. *Science* 323:133–38
55. Chaisson MJ, Pevzner PA. 2008. Short read fragment assembly of bacterial genomes. *Genome Res.* 18:324–30
56. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. 1990. Basic local alignment search tool. *J. Mol. Biol.* 215:403–10
57. Weiner P. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pp. 1–11. Washington, DC: IEEE Comput. Soc.
58. Giegerich R, Kurtz S. 1997. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. *Algorithmica* 19:331–53

59. Manber U, Myers G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22:935–48
60. Abouelhoda MI, Kurtz S, Ohlebusch E. 2004. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms* 2:53–86
61. Grossi R, Gupta A, Vitter JS. 2003. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'03*, pp. 841–50. Philadelphia: Soc. Indust. Appl. Math.
62. Apostolico A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, ed. A Apostolico, Z Galil, pp. 85–96. Berlin: Springer-Verlag
63. Ferragina P, Manzini G. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 390–98. Los Alamitos, CA: IEEE Comput. Soc.
64. Burrows M, Wheeler DJ. 1994. *A block sorting lossless data compression algorithm*. Tech. Rep. 124, Digit. Syst. Res. Cent., Palo Alto, CA
65. Ferragina P, Manzini G, Mäkinen V, Navarro G. 2007. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* 3:20
66. Bentley JL, Sleator DD, Tarjan RE, Wei VK. 1986. A locally adaptive data compression scheme. *Commun. ACM* 29:320–30
67. Ferragina P, Luccio F, Manzini G, Muthukrishnan S. 2009. Compressing and indexing labeled trees, with applications. *J. ACM* 57:4
68. Sirén J, Välimäki N, Mäkinen V. 2014. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 11:375–88
69. Jacobson GJ. 1988. *Succinct static data structures*. PhD Thesis, Carnegie Mellon Univ., Pittsburgh, PA
70. Bowe A, Onodera T, Sadakane K, Shibuya T. 2012. Succinct de Bruijn graphs. In *WABI 2012: Algorithms in Bioinformatics*, ed. B Raphael, J Tang, pp. 225–35. Berlin: Springer-Verlag
71. Camacho C, Coulouris G, Avagyan V, Ma N, Papadopoulos J, et al. 2009. BLAST+: architecture and applications. *BMC Bioinform.* 10:421
72. Simpson JT, Durbin R. 2010. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics* 26:i367–73
73. Ferragina P, Gagie T, Manzini G. 2012. Lightweight data indexing and compression in external memory. *Algorithmica* 63:707–30
74. Myers EW. 2005. The fragment assembly string graph. *Bioinformatics* 21:ii79–85
75. Birol I, Jackman SD, Nielsen CB, Qian JQ, Varhol R, et al. 2009. De novo transcriptome assembly with ABySS. *Bioinformatics* 25:2872–77
76. Indyk P, Motwani R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98*, pp. 604–13. New York: Assoc. Comput. Mach.
77. Wang J, Shen HT, Song J, Ji J. 2014. Hashing for similarity search: a survey. arXiv:1408.2927 [cs.DS]
78. Li P, Owen A, Zhang CH. 2012. One permutation hashing for efficient search and learning. arXiv:1208.1259 [cs.LG]
79. Bar-Yossef Z, Jayram TS, Kumar R, Sivakumar D, Trevisan L. 2002. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, ed. JDP Rolim, SP Vadhan, pp. 1–10. Berlin: Springer-Verlag
80. Indyk P, Woodruff D. 2003. Tight lower bounds for the distinct elements problem. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, 2003*, pp. 283–88. New York: IEEE
81. Kane DM, Nelson J, Woodruff DP. 2010. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '10*, pp. 41–52. New York: Assoc. Comput. Mach.
82. Giroire F. 2009. Order statistics and estimating cardinalities of massive data sets. *Discrete Appl. Math.* 157:406–27
83. Flajolet P, Fusy É, Gandouet O, Meunier F. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, pp. 127–46. Nancy, Fr.: Discret. Math. Theor. Comput. Sci.

84. Heule S, Nunkesser M, Hall A. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 683–92. New York: Assoc. Comput. Mach.
85. Cormode G, Muthukrishnan S. 2005. An improved data stream summary: the Count-Min sketch and its applications. *J. Algorithms* 55:58–75
86. Alon N, Matias Y, Szegedy M. 1999. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* 58:137–47
87. Woodruff D. 2004. Optimal space lower bounds for all frequency moments. In *SODA '04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 167–75. Philadelphia: Soc. Indust. Appl. Math.
88. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. 2014. Parallel de Bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 437–48. Los Alamitos, CA: IEEE Comput. Soc.
89. Melsted P, Halldórsson BV. 2014. KmerStream: streaming algorithms for *k*-mer abundance estimation. *Bioinformatics* 30:3541–47
90. Bloom BH. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13:422–26
91. Broder A, Mitzenmacher M. 2004. Network applications of Bloom filters: a survey. *Int. Math.* 1:485–509
92. Fan L, Cao P, Almeida J, Broder AZ. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking* 8:281–93
93. Cohen S, Matias Y. 2003. Spectral bloom filters. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 241–52. New York: Assoc. Comput. Mach.
94. Salikhov K, Sacomoto G, Kucherov G. 2014. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms Mol. Biol.* 9:2
95. Rozov R, Shamir R, Halperin E. 2014. Fast lossless compression via cascading Bloom filters. *BMC Bioinform.* 15:S7
96. Crainiceanu A. 2013. Bloofi: a hierarchical Bloom filter index with applications to distributed data provenance. In *Cloud-I '13: Proceedings of the 2nd International Workshop on Cloud Intelligence*, Pap. 4. New York: Assoc. Comput. Mach.
97. Crainiceanu A, Lemire D. 2015. Multidimensional Bloom filters. *Inform. Syst.* 54:311–24
98. Sun C, Harris RS, Chikhi R, Medvedev P. 2018. AllSome sequence Bloom trees. *J. Comput. Biol.* 25:467–79
99. Bender MA, Farach-Colton M, Johnson R, Kraner R, Kuszmaul BC, et al. 2012. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endow.* 5:1627–37
100. Knuth DE. 1998. *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Reading, MA: Addison–Wesley. 2nd ed.
101. Pandey P, Bender MA, Johnson R, Patro R. 2017. A general-purpose counting filter: making every bit count. In *SIGMOD '17: Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 775–87. New York: Assoc. Comput. Mach.
102. Raman R, Raman V, Rao SS. 2002. Succinct indexable dictionaries with applications to encoding *k*-ary trees and multisets. In *SODA '02: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 233–42. Philadelphia: Soc. Indust. Appl. Math.
103. Yu Y, Liu J, Liu X, Zhang Y, Magner E, et al. 2018. SeqOthello: Querying RNA-seq experiments at scale. *Genome Biol.* 19:167–80
104. Zerbino DR, Birney E. 2008. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* 18:821–29
105. Chikhi R, Rizk G. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol.* 8:22
106. Pell J, Hintze A, Canino-Koning R, Howe A, Tiedje JM, Brown CT. 2012. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *PNAS* 109:13272–77
107. Pandey P, Bender MA, Johnson R, Patro R. 2017. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33:i133–41

108. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. 2004. Reducing storage requirements for biological sequence comparison. *Bioinformatics* 20:3363–69
109. Schleimer S, Wilkerson DS, Aiken A. 2003. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 76–85. New York: Assoc. Comput. Mach.
110. Roberts M, Hunt BR, Yorke JA, Bolanos RA, Delcher AL. 2004. A preprocessor for shotgun assembly of large genomes. *J. Comput. Biol.* 11:734–52
111. Orenstein Y, Pellow D, Marçais G, Shamir R, Kingsford C. 2016. Compact universal *k*-mer hitting sets. In *WABI 2016: Algorithms in Bioinformatics*, ed. M Frith, CNS Pedersen, pp. 257–68. Cham, Switz.: Springer Int.
112. Marçais G, Pellow D, Bork D, Orenstein Y, Shamir R, Kingsford C. 2017. Improving the performance of minimizers and winnowing schemes. *Bioinformatics* 33:i110–17
113. Marçais G, DeBlasio D, Kingsford C. 2018. Asymptotically optimal minimizers schemes. *Bioinformatics* 34:i13–22
114. Myers EW, Sutton GG, Delcher AL, Dew IM, Fasulo DP, et al. 2000. A whole-genome assembly of *Drosophila*. *Science* 287:2196–204
115. Pérez N, Gutierrez M, Vera N. 2016. Computational performance assessment of k-mer counting algorithms. *J. Comput. Biol.* 23:248–55
116. Erbert M, Rechner S, Müller-Hannemann M. 2017. Gerbil: a fast and memory-efficient *k*-mer counter with GPU-support. *Algorithms Mol. Biol.* 12:9