Mismatch-tolerant, alignment-free sequence classification using multiple spaced seeds and multiindex Bloom filters

Justin Chu[a,1], Hamid Mohamadi[a], Emre Erhan[a], Jeffery Tse[a], Readman Chiu[a], Sarah Yeo[a], and Inanc Birol[a,b,1]

[a]Canada's Michael Smith Genome Sciences Centre, BC Cancer, Vancouver, BC V5Z 4S6, Canada; and [b]Department of Medical Genetics, University of British Columbia, Vancouver, BC V6T 1Z4, Canada

Edited by David L. Donoho, Stanford University, Stanford, CA, and approved June 12, 2020 (received for review March 1, 2019)

Alignment-free classification tools have enabled high-throughput processing of sequencing data in many bioinformatics analysis pipelines primarily due to their computational efficiency. Originally *k*-mer based, such tools often lack sensitivity when faced with sequencing errors and polymorphisms. In response, some tools have been augmented with spaced seeds, which are capable of tolerating mismatches. However, spaced seeds have seen little practical use in classification because they bring increased computational and memory costs compared to methods that use *k*-mers. These limitations have also caused the design and length of practical spaced seeds to be constrained, since storing spaced seeds can be costly. To address these challenges, we have designed a probabilistic data structure called a multiindex Bloom Filter (miBF), which can store multiple spaced seed sequences with a low memory cost that remains static regardless of seed length or seed design. We formalize how to minimize the false-positive rate of miBFs when classifying sequences from multiple targets or references. Available within BioBloom Tools, we illustrate the utility of miBF in two use cases: read-binning for targeted assembly, and taxonomic read assignment. In our benchmarks, an analysis pipeline based on miBF shows higher sensitivity and specificity for read-binning than sequence alignment-based methods, also executing in less time. Similarly, for taxonomic classification, miBF enables higher sensitivity than a conventional spaced seed-based approach, while using half the memory and an order of magnitude less computational time.

probabilistic data structures | spaced seeds | sequence classification | Bloom filters | alignment-free

In computational biology, sequence classification is a common task with many applications such as contamination screening (1), pathogen detection (2–4), metagenomics (3–5), and targeted assembly from shotgun sequence data (6, 7). Although this task can be carried out via sequence alignment (8), not all use cases require information on exact genomic coordinates, and sequence alignment algorithms perform more computation than necessary in such use cases. As the scale of modern datasets (both of the query and the reference sequences) grew rapidly, it has spurred the development of faster alignment-free, hash-based similarity methods (5). Here, we introduce a probabilistic data structure based on the Bloom filter (BF) (9), called multiindex Bloom filter (miBF). It implicitly stores data to reduce memory usage and employs multiple spaced seeds to represent sequences to better handle sequence polymorphisms and errors compared to other hash-based sequence classification methods.

The most common hash-based, alignment-free indexing methods are *k*-mer based. These methods work by breaking a reference sequence into subsequences of length k base pairs, and indexing them (often in a hash table). To query sequences, a given query is also broken into *k*-mers and interrogated against the index in search of shared *k*-mers. If a significant number of *k*-mers are found in this search, then the query is assigned to the reference (i.e., classified). In such applications, *k*-mers must be long enough so that they are unlikely to be common among multiple indexed targets, especially if there is substantial sequence similarity between targets. However, *k*-mers cannot compensate for minor differences between references and queries, such as single-nucleotide variations. These limitations have motivated us to use spaced seeds (10) [also called gapped q-grams (11)].

Spaced seeds are a modification to the standard *k*-mers, where some wildcard positions are added to allow for approximate sequence matching. They were originally proposed in PatternHunter in 2002 (10) and have been used since to improve the sensitivity and specificity of homology search algorithms (12–16). It has been illustrated that employing multiple spaced seeds together can increase the sensitivity of homology searches (17). They have also been used in metagenomics studies to improve the sensitivity of sequence classification (18, 19).

Probabilistic data structures are a class of data structures that focus on representing data approximately; consequently, query operations can sometimes produce false positives. The use of probabilistic data structures in bioinformatics has expanded in recent years, owing to their low memory usage and speed. To control the false-positive rates (FPRs) of these data structures, users can adjust the memory they use or the number of operations they perform. For example, in BF (9), false positives are reduced by lowering the filter occupancy (increasing the memory

Significance

The sustained growth in data throughput in DNA sequencing platforms surpasses the capacity growth in computing infrastructure. For many biological studies, faster alignment algorithms alleviated some of the computational burden, although not all genomics projects require alignments. Accordingly, recent alignment-free methods are enabling a range of tasks from transcript expression analysis, to metagenome characterization, to local de novo assembly. Although faster than alignment-based methods, these methods are often limited in their sensitivity and memory requirements. We demonstrate that by using spaced-seeds and probabilistic data structures, the sensitivity of alignment-free classification methods can be improved, with memory requirements independent of the seed design and linear to the indexed target size. Furthermore, we showcase the scalable parallelism of this approach.

Author contributions: J.C., E.E., R.C., and I.B. designed research; J.C., E.E., J.T., R.C., and S.Y. performed research; J.C., H.M., E.E., and J.T. contributed new reagents/analytic tools; J.C., E.E., and S.Y. analyzed data; and J.C. and I.B. wrote the paper.

The authors declare no competing interest.

This article is a PNAS Direct Submission.

Data deposition: The tool source code is available at GitHub (https://github.com/bcgsc/biobloom).

[1]To whom correspondence may be addressed. Email: cjustin@bcgsc.ca or ibirol@bcgsc.ca.

This article contains supporting information online at https://www.pnas.org/lookup/suppl/doi:10.1073/pnas.1903436117/-/DCSupplemental.

First published July 8, 2020.

usage) or by increasing the number of hash functions used per element exploiting the joint probability of multiple independent events (increasing the number of operations performed).

We note that, in bioinformatics applications, typically the data being indexed have certain aspects that may be exploited but are often overlooked. For instance, current sequence analysis methods that utilize probabilistic data structures for key–value associations consider every inserted key as an independent event (20, 21), assigning a single FPR for each key query. However, unlike in many other applications of these data structures in other computer science domains, in bioinformatics applications, multiple keys may relate to a decomposition of the same sequence and are thus not independent (22).

The use of BFs for sequence classification was first introduced to the field by the tool Fast and Accurate Classification of Sequences (23). Later, BioBloom Tools (BBT) (1) offered heuristics to optimize the runtime and reduce the effect of false positives. Although BBT proved effective when using a small number of references, classifying queried sequences into multiple references requires the use of a BF for each reference, resulting in an $O(n)$ time complexity per query, where n is the number of references. Here, we have extended the functionality of BFs for the classification against multiple references, providing key–value associations, through a data structure called a miBF. While miBF shares similarities with other BF extensions, such as Bloomier filters (24), sequence Bloom trees (25), quotient filters (26, 27), quasi-dictionary (21), Pufferfish (28), Othello (20, 29), and interleaved BFs (30), miBF has properties that allow it to synergize with spaced seeds. See *SI Appendix* for comparisons of the time and memory complexities of these data structures.

At suggested parameterization, miBF requires around 20 bits per key per hash, assuming 16-bit values to enumerate the references. Look-up time of miBF is constant and requires at most two cache misses per key lookup. Unlike other BF-related data structures, the FPR for a single key lookup in miBF may differ depending on the classification result, in addition to the parameters of the miBF and length of the sequence being queried.

We implemented a generic sequence classifier using the miBF and present its utility in two use cases: read-binning for targeted assembly, and metagenomic classification. We compared the performance of the miBF-based classifier against state-of-the-art tools BWA-MEM and CLARK/CLARK-S for the two use cases, respectively. The data structure and the classifier used here are provided through our software repository at https://github.com/bcgsc/biobloom.

Results

Filtering Reads for Targeted Assembly. Targeted assembly can improve the throughput and reduce the complexity of assembly for applications such as clinical diagnostics for structural variants or other mutations. A typical procedure when performing targeted assembly is the extraction, or binning, of sequencing, reads in the target loci, before using the reads for de novo assembly. This binning can be done via alignment or sequence classification since exact genomic coordinates are not necessary for this application. Here, we compare the binning of reads with BWA-MEM (31) and BBT with miBF using a set of simulated reads. For this experiment, we simulated Illumina reads to a depth of 100× (2,303,019 read pairs) using pIRS (v1.1.1) (32) from genomic loci around 580 COSMIC (v77) genes (33) and an equal number of non-COSMIC genes randomly selected from RefSeq (34). We indexed the set of 580 genes into a BWA-MEM FM-index and a BBT miBF using a set of four spaced seeds (*SI Appendix*, Note S1).

Compared to BWA-MEM, BBT obtained a higher overall sensitivity (99.996% vs. 99.801%) and lower overall FPR (0.400% vs. 3.757%). On a per-gene basis, BBT outperforms BWA-MEM in terms of the F1 score (Fig. 1 and *SI Appendix*, Fig. S1). When the reads by both tools were assembled using ABySS (35), a Quast analysis (36) on the target gene list showed comparable assembly results (*SI Appendix*, Table S1), and no misassemblies. However, there were a few unmapped sequences assembled in the BWA-MEM binned set, suggesting some off-target sequences were assembled (*SI Appendix*, Table S1). We compared the runtime of each tool, finding that BBT runs at least 2× faster than BWA-MEM, and scales better on more threads (*SI Appendix*, Fig. S2). Memory usage of the classification stage of BBT on this set of 580 genes was 20 MB. Tests were performed on a machine with Intel Xeon E7-8867 2.5 GHz.

We also compared BBT and BWA-MEM for their memory usage and time of indexing, but because the indexed set of COSMIC genes was small, it does not offer a fair use case for indexing. Instead, to compare the scalability of the two tools, we indexed a 3.5G fasta file consisting of ∼1,000 bacterial sequences, using the same seed design for the miBF and default parameters for both tools. BWA-MEM indexing algorithm does not offer parallelization, and it took 1.5 h to index in the entire file. Using a single thread BBT took 6.7 h to index the same file, which dropped to only 0.7 h using 16 threads. BWA-MEM used only 5 GB of memory to index this file, whereas BBT used 33 GB.

Metagenomic Classification. Although BBT is a generic classification tool, when given the proper reference sequences, it can be used as the workhorse for a metagenomics classification pipeline. To demonstrate this, we compared BBT with CLARK (4) and CLARK-S (19) (the spaced seed variant of CLARK). We compared CLARK-S because it is the only metagenomic classification tool that we know of that supports multiple spaced seeds to improve classification sensitivity. We also compared the method to CLARK, the predecessor to CLARK-S, because it is well characterized against other tools (37).

We first generated a bacterial and viral genome database (constructed May 2018) for both CLARK-S and CLARK and used the same reference sequences and taxon IDs to construct the miBF (*SI Appendix*, Note S2). To index the genomes, CLARK took 24.0 h, CLARK-S took 24.5 h, and BBT took 8.5 h. We requested 64 threads for each tool, but CLARK/CLARK-S generally did not use more than 1 CPU at a time, while BBT used around 13 CPUs on average, suggesting that CLARK/CLARK-S have room to optimize parallelism for indexing. Memory usage when indexing was 170 GB for CLARK, 206 GB for CLARK-S, and 190 GB for BBT.

There are differences between the miBF index and CLARK/CLARK-S databases beyond our implicit representation of sequences. First, unlike CLARK and CLARK-S, in miBF, we do not remove sequences shared between different taxa but distribute them between taxa, and if repetitive they will be marked as "saturated" (*Methods*). Also, although CLARK-S and BBT both use multiple spaced seeds, miBF does not use the same set of seeds as CLARK-S because of our restrictions on seed designs (*Methods*). In addition, because our seed design does not affect memory usage of the miBF, we were also free to use longer seeds (*SI Appendix*, Note S2).

In these benchmarks, we used the simulated metagenomic datasets from the CLARK-S publication (19). However, because the National Center for Biotechnology Information databases have changed since the original CLARK-S publication, we had to omit reads simulated from genomes that no longer have a corresponding species taxon. Nevertheless, since we omit the same reads in all runs and use the same reference sequences, our results still yield a fair comparison. We generated two sets of simulated reads as outlined in the CLARK-S paper; the difference between the "default" and "unambiguous" sets is the unambiguous set does not have reads with all 32-mers shared between any two taxa IDs, but we note that, because the database has changed, this distinction may no longer hold completely true. The default datasets are

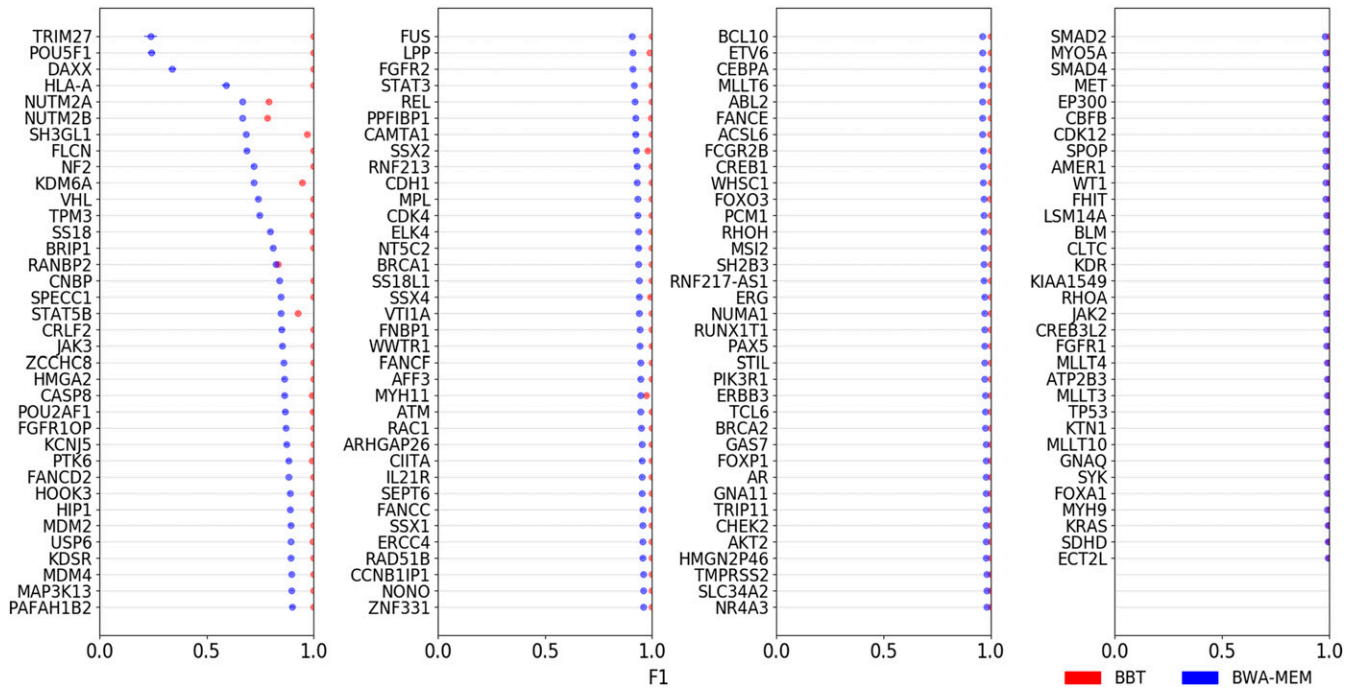

Fig. 1. Per-gene comparison of classification performance by BBT vs. BWA-MEM. F1 scores for both methods are calculated for each gene and plotted on the same horizontal line. Only genes with F1 scores less than 99.9% are shown here (for full set, see *SI Appendix*, Fig. S1). The scale on the *x* axis for BWA-MEM on the right is reversed for easy visual comparison, such that higher scores for both methods localize in the middle.

more representative of real datasets, and the unambiguous datasets are more idealized for CLARK and CLARK-S.

CLARK only produces a single best match, whereas CLARK-S also produces a secondary hit. Like CLARK-S, BBT produces secondary hits, but unlike CLARK-S, BBT can report more than one secondary hit. Thus, we compared the performance of all three tools using only the best hits (Fig. 2*A*) and compared the performance of CLARK-S and BBT using multiple hits (Fig. 2*B*). In this test, BBT reports only 5.4% of the reads to have multihits, and of that, a majority (75.2%) hit only two targets (Fig. 2*B*). CLARK-S provides a secondary hit 40.7% of the time, thus suffering in precision when considering multiple hits.

For best hits, as expected, CLARK-S has higher sensitivity than CLARK in almost all cases, reproducing the results found in the original CLARK-S paper (Fig. 2*A* and *SI Appendix*, Tables S2 and S3). In our tests, BBT has the highest sensitivity, allowing it to yield the highest F1 score in all but one case (Unambiguous simBA-525 dataset). In addition, we tested BBT with parameterizations to increase precision by filtering ambiguous elements (Fig. 2 *C* and *D* and *SI Appendix*, Note S2). In most datasets, BBT outperformed CLARK and CLARK-S in sensitivity at the same precision, with the exception of the Unambiguous simBA-525 dataset and Default Sio50 dataset.

Included with the CLARK-S datasets are three negative control datasets totaling in 3 million 100-bp pair reads. Unexpectedly, some of the reads in the negative control datasets had mapped in both CLARK-S (six reads) and BBT (one read). This contradicts with the results reported in the original CLARK-S paper where no reads were mapped in any of the negative controls. In BBT, we expected no false positives either because the minimum FPR parameter (-s) was specified to be less than 10^{-10} (default). We hypothesize this may be due to the difference in the databases, possibly some of the new reference genomes having sequences similar to those found in the negative control. Overall, we think this should not be a cause for concern, representing a FPR in parts per million.

We tested the runtime of each method at a differing number of threads (Fig. 2*E*). We show that CLARK is the fastest tool, followed by BBT and finally CLARK-S. This trend holds when more threads are used, although BBT seems to scale better than CLARK, rivalling it at a higher number of threads. Execution times of BBT and CLARK stay within an order of magnitude of each other, while CLARK-S takes an order of magnitude longer to run. In our test, CLARK used 87 GB, BBT used 89 GB, and CLARK-S used 175 GB. Database loading speed was not included in our benchmarks and was comparable between the three methods.

Discussion

We have presented the miBF, a probabilistic sequence classification data structure that can perform multiple key–value associations and synergizes well with multiple spaced seeds. Like BFs, the memory usage of miBF does not depend on the size of the hashed elements. With a BF, querying for the set of origin between multiple reference sets requires the construction and use of multiple BFs, leading to O(n) time complexity when interrogating against n reference sets/filters. In contrast, querying for the set of origin for an element in miBF requires only one instance of the data structure, and is performed in constant time. Packaged and implemented in BBT, we have shown it to be a practical sequence classification tool.

To optimize the available memory for a target FPR, BFs can use multiple hash functions to describe a given k-mer. We modified this concept in miBF: Instead of using multiple hash values for the same k-mer, we hash multiple spaced seed templates across it. Naively, one may simply insert each spaced seed as its own element (using multiple hash functions for each insertion); yet, since the seeds describing the same k-mer are dependent, we can instead use a set of spaced seeds in the place of multiple hash functions. Also, by allowing for some spaced seeds to miss, we can tolerate mismatches when classifying sequences.

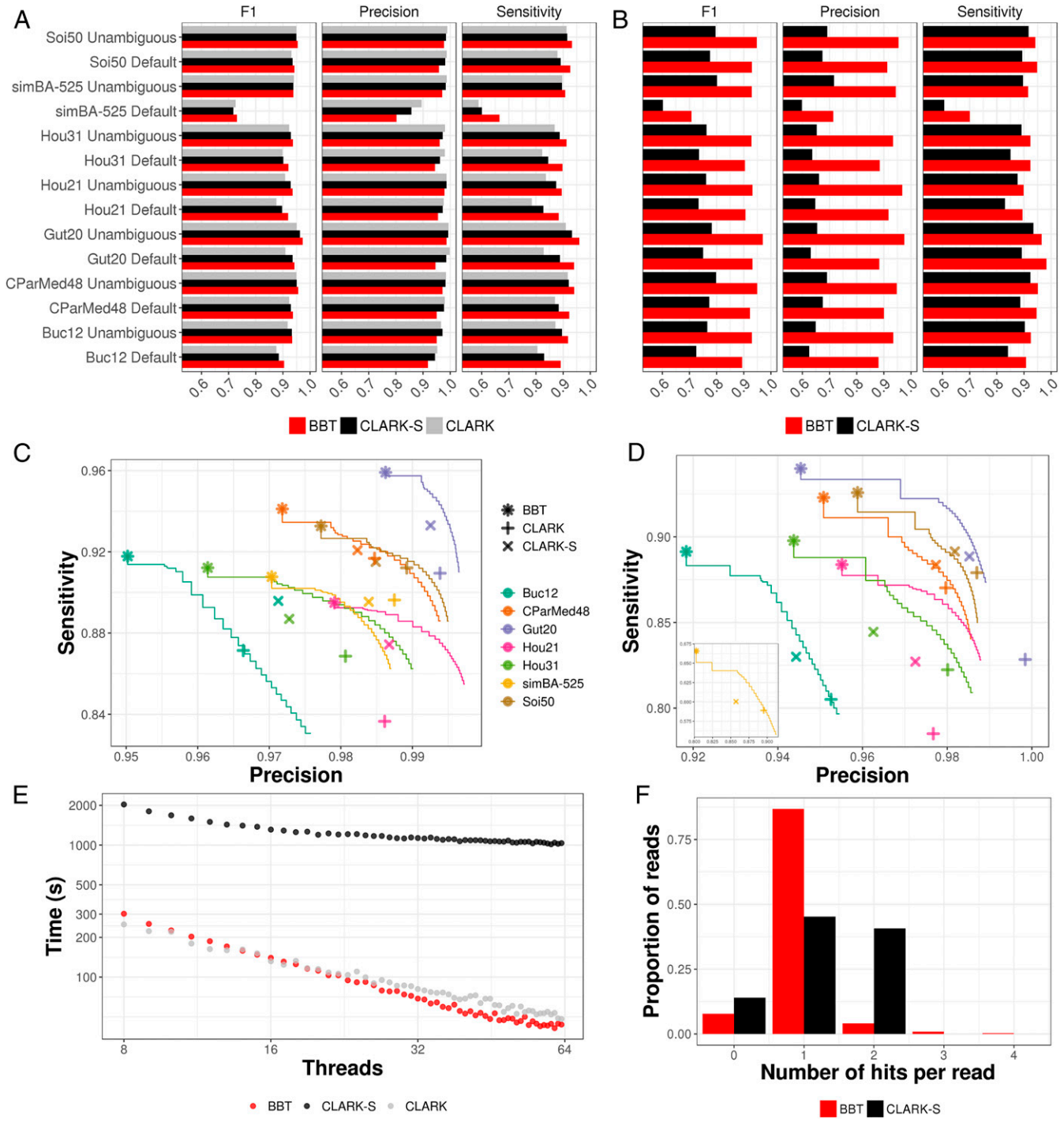

Fig. 2. Comparison of CLARK, CLARK-S, and BBT. (*A*) Precision and sensitivity considering only the best hit of a classification. (*B*) Precision and sensitivity considering all multimaps. CLARK is omitted as it produces a single hit. (*C*) Sensitivity vs. precision plot considering best hits on the unambiguous dataset. Lines indicate BBT runs parameterized to yield a higher precision (*SI Appendix*, Note S2). (*D*) Sensitivity vs. precision plot considering best hits on the default dataset. (*E*) Runtime comparison of CLARK, CLARK-S, and BBT at 8 to 62 threads. Both axes are in log scale, and under the situation of perfect scaling, the trend should follow a linear slope. (*F*) Number of hits per record in all CLARK-S datasets for BBT and CLARK-S. Again, CLARK is not included because it does not multimap sequences.

Seed size has no impact on memory usage on this data structure, allowing explorations for specialized, highly sensitive, and specific spaced seed designs. Optimal seed design is an NP-hard problem (38), and although faster approximations exist (39), the problem remains difficult as our seeds can be of any length. In addition to the seed length, the performance of designed seeds would be a function of the sequencing error rate, homology detection tolerance, and mutation/error types. Computationally, there are methods to hash multiple spaced seeds more efficiently (40), and some seed designs can be hashed more efficiently than others.

We investigated the impact of seed design on classification performance by randomly sampling the universal set of spaced

seeds of a given length. In practice, using completely randomly designed seeds, it was difficult to create poorly performing designs. We used a generative Markov process (41) that purposely created seeds with high and low Shannon entropy (42). We tried combinations of these seeds in sets of five seeds. Our results show that seed design does matter, but even poorly generated spaced seeds tend to perform better compared to *k*-mers (Fig. 3), suggesting one can expect gains on sensitivity relative to *k*-mers without extensive work on multiple spaced seed design. Thus, the seeds used throughout the paper were randomly generated with a script provided as part of BBT after picking a weight and seed length (*Methods*). As seed design improves, we expect the performance of our tool to improve.

To facilitate key–value lookups, miBF stores an index for each hashed key. These indexes represent a one-to-one mapping of the IDs in a list of references onto integer numbers. We denote a particular instance of the index by *i*. Due to shared sequences among references or hash collisions, a loss of key–value associations can occur. If enough collisions occur such that key–value association information for a query is lost, we denote this element as saturated. To record saturation, we reserve and set one bit (the saturation bit) in the integer used to store the ID. For instance, for a miBF with five seeds, if a query sequence is present in more than five references, the miBF can only record up to five indexes; the saturation bit will be set in this case and indicates that the actual number of references a hashed key may be associated with may be more than five.

We have designed miBF for the use case of classifying a sequence by interrogating a collection of its subsequences, or "frames," of fixed length, typically using a sliding window. This allows us to classify a query sequence even when some of its frames are flagged to have saturated representations.

In practice, the rate of saturation can be mitigated to, say 1% (depending on the number and relative repetitiveness of references), by using only four spaced seeds and a 50% BF occupancy (*SI Appendix*, Fig. S4). Also, although statistically unbiased, the representative counts can vary between IDs, but this variance can be minimized by using more spaced seeds (*SI Appendix*, Fig. S5).

When querying, we determine whether a sequence is a false positive by using the frame counts of reported associations (*SI Appendix*, Note S4). We have formulated a model for calculating the FPR given a sequence (rather than a single element query), where FPR is a function of the retrieved ID's frequency in the filter. Interestingly, and somewhat counterintuitively, this means miBFs with more IDs will overall have lower FPRs, especially when references have fewer shared sequences. Our FPR formulation is also a function of the length of the input sequence and is more robust than simpler measures based on the proportion of hits across the frames of an input sequence. Finally, our FPR has a subtractive formula, allowing for early classification termination when a target threshold is reached.

Frame counts of reported associations are generally a reliable measure for ranking multimatches but can be distorted if the query sequence contains subsequences shared among multiple references. When disambiguating multimatches, we found the nonsaturated frame counts (*SI Appendix*, Note S4) to be a better measure.

We showcased our classification tool BBT using miBF in two use cases. The first use case was the recruitment of reads for targeted assembly. In our tests, BBT had better sensitivity and lower FPR in comparison to BWA-MEM while executing in less time. We note that the use of spaced seeds affords a certain level of specificity that allows BBT to index only the target of interest. In contrast, BWA-MEM needs to index a more comprehensive reference to prevent off-target classification. The overall specificity of BBT was higher than that of BWA-MEM, but we note that for one target gene BWA-MEM outperformed BBT because BWA-MEM had access to the entire gene set. This suggests that a spaced seed approach may be superior in terms of overall specificity when using incomplete or missing reference sequences. The practical speed and favorable sensitivity of miBFs have prompted the use of BBT in the targeted transcriptome assembly pipeline called TAP (43) and may be a good fit for other pipelines performing read binning.

The second use case was the classification of metagenomic sequences to the species level. Under default parameterization, BBT showed higher sensitivity than both CLARK-S and CLARK. To illustrate that the gains to sensitivity were not necessarily at the cost of precision, we also ran BBT with parameters to allow for more precise classifications, essentially by filtering ambiguous matches from the output. Parametric curves of sensitivity vs. precision indicate that BBT shows generally higher sensitivity at the same precision to CLARK and CLARK-S. BBT's sensitivity gains over CLARK-S were likely due to the use of slightly lower weight seeds in BBT with more seeds (four in BBT vs. three in CLARK-S), and because in miBF we do not filter out sequences that are shared between references. Also, BBT compensates for the decrease in seed weights by using longer seeds (42 bp by default, longer than what CLARK-S can currently use) to achieve higher specificity.

Both CLARK-S and BBT can provide multimatches. Although BBT may get the best hit incorrect in some cases, we observe that one of the secondary hits is often the correct classification. When the multimatches were considered, BBT still outperformed CLARK-S in sensitivity, and notably also had a much higher relative precision. However, this comparison to CLARK-S may be unfair since CLARK-S produces alternative hits liberally (penalizing precision), whereas BBT only produces multihits when there is high ambiguity. In most cases, CLARK-S will often get the best hit correct, whereas BBT will have fewer reads with multihits, although these classifications would be much more likely to be true multimaps.

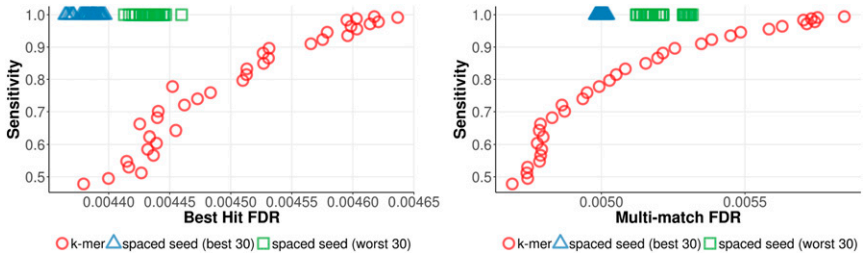

Fig. 3. Sensitivity vs. FDR plots for best hits (*Left*) and multimatches (*Right*) investigating sets of multiple spaced seeds (60 designs) against *k*-mers (20 to 60) on classification on a miBF generated on 580 genes from the COSMIC database using reads 2 × 150-bp reads simulated with a 0.4% error rate on the same set of genes. Each set of spaced seeds has five seeds with the same weight of 20 and the same length of 60. The *k*-mers used five hash functions to make the miBFs comparable.

The runtime of BBT remained around two times slower than CLARK and it generally scaled better than CLARK when more threads were used. When parameterized for higher specificity BBT ran slower. Due to the implicit representation of spaced seeds, BBT used half the memory compared to CLARK-S and a similar amount of memory compared to CLARK. The runtime of CLARK-S was more than an order magnitude slower than CLARK and BBT, suggesting that the computation of multiple spaced seeds can be quite expensive if not carefully optimized.

In conclusion, in addition to illustrating a tool with practical value for sequence classification, we introduce a theoretical framework in using multiple spaced seeds represented by a miBF. The concepts described would be of interest to a wider audience, including researchers in various fields of computer science–data structures in particular. Ideas and concepts toward minimizing FPRs in massive volumes of queries would be of particular interest to those studying probabilistic data structures, which power the internet.

Methods

Multiple Spaced Seeds. A spaced seed is a template of zeros and ones, where ones indicate required match positions on a sequence, and zeros indicate wild-card positions where a match or no-match is immaterial. The number of ones in a spaced seed is defined to be its weight. Just like using a *k*-mer spectrum of a sequence, where subsequences of length *k* are extracted from a longer sequence, we consider "frames" of a fixed length across a sequence and hash multiple spaced seeds for the sequence in each frame. We impose no restriction on length or weight for spaced seeds. However, we require each seed to either have a mirrored template with another seed or be palindromic. This allows us to save memory by storing each seed only once and not both forward and reverse complements, and it is analogous to storing canonical *k*-mers (comparing the forward and reverse complement of a *k*-mer and consistently using one).

The seeds used in our experiments were randomly generated subject to the following three conditions. We required that 1) they have 50% occupancy; 2) in a set of four seeds, there are exactly two seeds with a wildcard in each position; and 3) two of the seeds are mirrored templates of the other two seeds. This design ensures that if we have a single mismatch in a query sequence at least two of the seeds report hits.

We calculated the Shannon entropy of spaced seed patterns by breaking them down into subsequences (up to a word size) and used the relative frequencies of these words as their respective event probabilities (42). The seeds used in the read binning experiments were 80 bp in length, and have Shannon entropies of 2.953 and 2.958 bits (using a word size of 3), with mirrored templates having the same entropy. The seeds used in the metagenomic experiments were 42 bp in length and have Shannon entropies of 2.939 and 2.950 bits (using a word size of 3), again with mirrored templates having the same entropy.

miBF Structure. The miBF can be thought of as three separate arrays. The first is a BF bit array, storing the presence or absence of an element in the set (Fig. 4*A*). The next one is an array that stores the rank information of the bit array at specific intervals, counting the number of set bits prior to the current position; this allows for constant time rank information access (44) to positions on the bit array. The third is an array that stores integer identifiers for each element in the bit array. The rank array in conjunction with the ID array can be used to retrieve the integer identifiers for all set positions in the BF. To improve cache performance, the BF and rank arrays are interleaved into a single data vector (45) (Fig. 4*B*).

miBF Construction. To minimize miBF construction memory usage overhead, all data are streamed, but multiple threads can be used in each pass to improve runtime. Construction of the miBF consists of three passes through the sequence set being indexed (*SI Appendix*, Note S5). The first pass populates the BF, the second pass populates the ID array, and a final pass scans over all colliding IDs recovering key–value associations when possible and sets saturation bit in the ID array. The first pass is performed in parallel per element using atomic bitwise OR operations on the BF. In the second and third passes, elements can be updated concurrently in the data array by using compare-and-swap. In practice, we parallelize construction per index to more easily prevent duplicate elements per index from being inserted (important for preventing bias when dealing with collisions between indexes) and to enable more efficient hashing via ntHash (46). Duplicate elements within the same index are prevented by using a temporary hash table.

Due to shared sequences or hash collisions, inserted values into the ID array may collide, causing a loss of key–value association information. Although we allow for collisions to replace existing IDs in the data structure, we do so using reservoir sampling (47). Reservoir sampling allows for an equal chance of replacement of a reservoir of elements with the intention of obtaining a random sampling of elements from streaming data of indeterminate size. It works by adjusting the probability of replacement to be based on how many collisions for that location have been seen previously by keeping a count of how many elements have been seen in the stream. Thus, in our implementation, we use a temporary count vector of the same size as the data vector to record the number of times a given location in the data vector is considered for insertion (Fig. 5*A* and *SI Appendix*, Note S5).

In addition to allowing some replacement of collided IDs, we flag any loss of key–value information by using left most bit of our IDs (that we call a saturation bit) to indicate that for an element (an element is considered a set of spaced seeds to one sequence position). This is done in the final pass (Fig. 5*B* and *SI Appendix*, Note S5) and has the benefit of not destroying key–value associations that are used in other elements. Finally, if no representative seeds exist for that element, but a duplicate ID is found (Fig. 5*C* and *SI Appendix*, Note S5), we can recover it by inserting the value into one of the duplicate positions.

Generally, more spaced seeds would help decrease saturation (*SI Appendix*, Fig. S4) as well as the variance of each count (*SI Appendix*, Fig. S5). However, this would be at the cost of increased memory usage. Our construction method shows little sequence bias in practice.

BF size constructed is based on the number of base pairs in the reference, and miBF would create a filter with an approximate specified occupancy. Assuming uniformly distributed hash values, this should yield a filter at the specified occupancy or less. Also, unless there is substantial sequence duplication in the reference, the occupancy of the BF should not be drastically lower than the specified occupancy.

Sequence Classification. We perform classification in two stages. First, we decide if the matches to an ID are enough to indicate the read is not likely to be a false-positive match, determined by a preset minimum FPR (1 in 10^{10} by default). Then we rank the significant candidate IDs and select candidates that match strongly enough that they can be considered a multimatch. A lookup of the data structure is performed by hashing a set of spaced seeds in a frame, and confirming if the associated filter positions are occupied, and if they are, their rank values are used to retrieve the matching ID from the ID array. In this scheme, one can end up with multiple IDs for the query, with a high likelihood of one of them being a false positive. To query sequences reliably, we use frame matches within the same sequence to the same ID to reduce the effective FPR, because these matches should be independent events if they occur due to hash collisions. These frame matches can thus be used collectively to compute a joint probability that the sequence match is encountered by chance. This is similar to other methods shown to reduce the effective FPR in BFs (22), with the difference here being multiple IDs with different frequencies in the filter need to be considered.

The chance of a false positive depends on the length of the sequence being queried, the length of the seed used, the FPR of the BF, and the frequency of

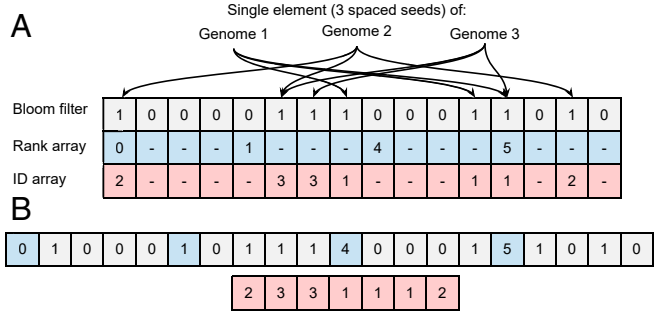

Fig. 4. (*A*) A visualization of the miBF data structure. Three tables are used to represent the miBF and how they are related. (*B*) The form of the data actually stored with an interleaved form of the bit vector. The interval for the rank array is much larger than shown here (4 vs. 512 bits), reducing its overhead to 64/512 = 0.125 bits per position.

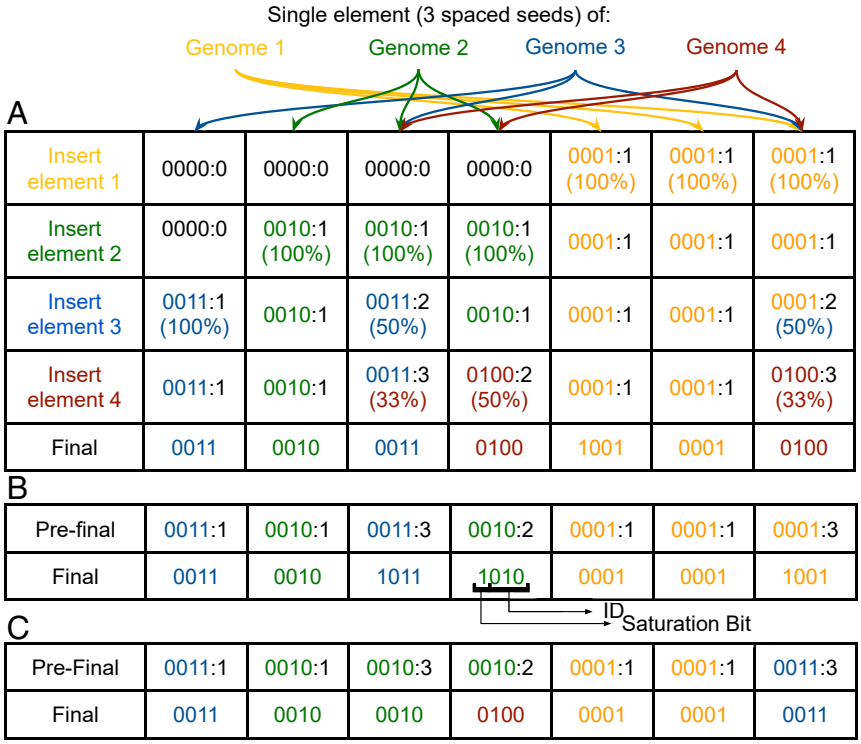

Fig. 5. An illustration of a miBF data vector construction using 3-bit IDs representing four references (genomes) of single elements. Values after the colons are the number of times an ID was considered to be inserted into that bucket, needed for reservoir sampling. The percentage in the parentheses represents the chance of insertion of a different element into that position if different from the cell above. (*A*) Insertions into miBF of different keys randomized with reservoir sampling. Before the final pass, each element has at least one representative spaced seed inserted so no changes are needed. (*B*) An example order of insertions causing saturation of some of the key–value association in the miBF. No replacement locations for element from genome 4 can be found, so the hashed values for the three seeds are saturated. (*C*) An example where an element from genome 4 is recovered in the final pass. Rather than saturation, an ID (from genome 2) that is found more than once is replaced by genome 4.

each ID in the ID array. When classifying a sequence, the FPR for each frame is a series of n independent Bernoulli trials (number of frames) so we can model the overall chance of a false positive using a binomial distribution. Our miBF FPR formulation is based on the BF FPR (9) as follows:

$$f = b^h = \sum_{x=h}^{h} \binom{h}{x} b^x (1-b)^{h-x},$$

where b is the occupancy of the BF and h is the number of spaced seeds (traditionally number of hash functions) used for a single frame in the sequence.

To tolerate for few sequence mismatches when classifying, we allow some spaced seeds in a frame to have BF misses. By default, we accept all but one seed in a frame to miss, but this can be changed ($-a$) to help decrease the FPR, at the cost of sensitivity. Thus, the formulation becomes the following:

$$f = \sum_{x=h-a}^{h} \binom{h}{x} b^x (1-b)^{h-x},$$

where a is the number of allowed misses for the set of spaced seeds in a frame. For a BF, the chance of falsely classifying a sequence is easily determined by computing the cumulative density function of the number of matches $m - 1$ and inverting it:

$$P(n, m) = 1 - \sum_{x=0}^{m-1} \binom{n}{x} (f)^x (1-f)^{n-x},$$

where n is the number of frames a query sequence has.

In miBFs, use of multiple indexes would help further reduce the per index FPR, although each index i represents an additional test. Before we compute the overall probability of an entire sequence for a given index i, we must first formulate the probability of falsely matching a frame of classification:

$$f_i = \sum_{x=h-a}^{h} \binom{h}{x} b^x (1-b)^{h-x} (1-(1-s_i)^x),$$

where s_i is the frequency of index i in the miBF data array, relative to the frequency of all indexes. Thus, the overall probability for false classification for index i is as follows:

$$P(n, i) = 1 - \sum_{x=0}^{m_i-1} \binom{n}{x} (f_i)^x (1-f_i)^{n-x},$$

where m_i is the number of frames reporting a match to index i.

Out of all of the tests for each index i, we then take the best candidate (lowest P value) and perform multiple test correction. In our implementation, we simply perform the Bonferroni correction (48):

$$P'(n, m_i) = N \times P(n, m_i).$$

We explored different corrections using simulated data (*SI Appendix*, Fig. S4) and showed that these correction methods generally result in similar corrected values for smaller critical values.

We filter in only matches that pass a minimum FPR threshold during classification. When the sequence classified is of a fixed length, we compute a fixed significant match threshold for each index by applying our Bonferroni-corrected critical P value with a quantile function (49). The classification will terminate early to improve execution time. We require a number of unambiguous matches (-r, default is 3) to terminate early. This heuristic has no effect on the FPR and only affects the accuracy of multimatches (*SI Appendix*, Note S3). Frame matches that include saturation bits are penalized depending on the extent of saturation in our implementation to improve the correctness of the classification (*SI Appendix*, Note S4).

Data Availability. All experimental data discussed in the paper were obtained from the CLARK-S publication (18). The tool source code is available at GitHub (https://github.com/bcgsc/biobloom).

ACKNOWLEDGMENTS. We thank Rachid Ounit for providing support in the proper use of CLARK and CLARK-S, as well as providing insight into understanding the results produced using these tools with the datasets from the CLARK-S publication. This work was supported by the National Human Genome Research Institute of the National Institutes of Health (R01HG007182), with additional support provided by Genome Canada and Genome British Columbia (281ANV). The content of this work is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health or other funding organizations.

1. J. Chu *et al*., BioBloom tools: Fast, accurate and memory-efficient host species sequence screening using Bloom filters. *Bioinformatics* **30**, 3402–3404 (2014).
2. A. D. Kostic *et al*., PathSeq: Software to identify or discover microbes by deep sequencing of human tissue. *Nat. Biotechnol.* **29**, 393–396 (2011).
3. D. E. Wood, S. L. Salzberg, Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biol.* **15**, R46 (2014).
4. R. Ounit, S. Wanamaker, T. J. Close, S. Lonardi, CLARK: Fast and accurate classification of metagenomic and genomic sequences using discriminative *k*-mers. *BMC Genomics* **16**, 236 (2015).
5. M. A. Peabody, T. Van Rossum, R. Lo, F. S. L. Brinkman, Evaluation of shotgun metagenomics sequence classification methods using in silico and in vitro simulated communities. *BMC Bioinformatics* **16**, 363 (2015).
6. R. L. Warren, R. A. Holt, Targeted assembly of short sequence reads. *PLoS One* **6**, e19816 (2011).
7. E. Kucuk *et al*., Kollector: Transcript-informed, targeted de novo assembly of gene loci. *Bioinformatics* **33**, 1782–1788 (2017).
8. A. L. Bazinet, M. P. Cummings, A comparative evaluation of sequence classification programs. *BMC Bioinformatics* **13**, 92 (2012).
9. B. H. Bloom, Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13**, 422–426 (1970).
10. B. Ma, J. Tromp, M. Li, PatternHunter: Faster and more sensitive homology search. *Bioinformatics* **18**, 440–445 (2002).
11. S. Burkhardt, J. Kärkkäinen, "One-gapped q-gram filters for Levenshtein distance" in *Annual Symposium on Combinatorial Pattern Matching*, A. Apostolico, M. Takeda, Eds. (Springer, 2002), pp. 225–234.
12. M. Ghandi, D. Lee, M. Mohammad-Noori, M. A. Beer, Enhanced regulatory sequence prediction using gapped k-mer features. *PLoS Comput. Biol.* **10**, e1003711 (2014).
13. L. Noe, D. E. K. Martin, A coverage criterion for spaced seeds and its applications to support vector machine string kernels and *k*-mer distances. *J. Comput. Biol.* **21**, 947–963 (2014).
14. L. Ilie, H. Mohamadi, G. B. Golding, W. F. Smyth, BOND: Basic OligoNucleotide design. *BMC Bioinformatics* **14**, 69 (2013).
15. G. Kucherov, L. Noé, M. Roytberg, A unifying framework for seed sensitivity and its application to subset seeds. *J. Bioinform. Comput. Biol.* **4**, 553–569 (2006).
16. L. Ilie, S. Ilie, A. M. Bigvand, SpEED: Fast computation of sensitive spaced seeds. *Bioinformatics* **27**, 2433–2434 (2011).
17. J. Qi, H. Luo, B. Hao, CVTree: A phylogenetic tree reconstruction tool based on whole genomes. *Nucleic Acids Res.* **32**, W45–W47 (2004).
18. K. Břinda, M. Sykulski, G. Kucherov, Spaced seeds improve *k*-mer-based metagenomic classification. *Bioinformatics* **31**, 3584–3592 (2015).
19. R. Ounit, S. Lonardi, Higher classification sensitivity of short metagenomic reads with CLARK-S. *Bioinformatics* **32**, 3823–3825 (2016).
20. X. Liu *et al*., A novel data structure to support ultra-fast taxonomic classification of metagenomic sequences with *k*-mer signatures. *Bioinformatics* **34**, 171–178 (2018).
21. C. Marchet, L. Lecompte, A. Limasset, L. Bittner, P. Peterlongo, A resource-frugal probabilistic dictionary and applications in bioinformatics. *Discrete Appl. Math.* **274**, 92–102 (2018).
22. D. Pellow, D. Filippova, C. Kingsford, Improving Bloom filter performance on sequence data using k-mer Bloom filters. *J. Comput. Biol.* **24**, 547–557 (2017).
23. H. Stranneheim *et al*., Classification of DNA sequences using Bloom filters. *Bioinformatics* **26**, 1595–1600 (2010).
24. B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables" in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, (Society for Industrial and Applied Mathematics, 2004), pp. 30–39.
25. B. Solomon, C. Kingsford, Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.* **34**, 300–302 (2016).
26. M. A. Bender *et al*., Don't thrash. *Proceedings VLDB Endowment* **5**, 1627–1637 (2012).
27. P. Pandey *et al*., Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Syst.* **7**, 201–207.e4 (2018).
28. F. Almodaresi, H. Sarkar, A. Srivastava, R. Patro, A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics* **34**, i169–i177 (2018).
29. Y. Yu, D. Belazzougui, C. Qian, Q. Zhang, Memory-efficient and ultra-fast network lookup and forwarding using Othello hashing. *IEEE/ACM Transactions on Networking* **26**, 1151–1164, 10.1109/TNET.2018.2820067 (2018).
30. T. H. Dadi *et al*., DREAM-Yara: An exact read mapper for very large databases with short update time. *Bioinformatics* **34**, i766–i772 (2018).
31. H. Li, Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997 (26 May 2013).
32. X. Hu *et al*., pIRS: Profile-based Illumina pair-end reads simulator. *Bioinformatics* **28**, 1533–1535 (2012).
33. S. A. Forbes *et al*., COSMIC: Somatic cancer genetics at high-resolution. *Nucleic Acids Res.* **45**, D777–D783 (2017).
34. N. A. O'Leary *et al*., Reference sequence (RefSeq) database at NCBI: Current status, taxonomic expansion, and functional annotation. *Nucleic Acids Res.* **44**, D733–D745 (2016).
35. S. D. Jackman *et al*., ABySS 2.0: Resource-efficient assembly of large genomes using a bloom filter. *Genome Res.* **27**, 768–777 (2017).
36. A. Gurevich, V. Saveliev, N. Vyahhi, G. Tesler, QUAST: Quality assessment tool for genome assemblies. *Bioinformatics* **29**, 1072–1075 (2013).
37. S. Lindgreen, K. L. Adair, P. P. Gardner, An evaluation of the accuracy and speed of metagenome analysis tools. *Sci. Rep.* **6**, 19233 (2016).
38. M. Li, B. Ma, L. Zhang, "Superiority and complexity of the spaced seeds" in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm–SODA '06*, (Society for Industrial and Applied Mathematics, Philadelphia, 2006), pp. 444–453.
39. L. Hahn, C.-A. Leimeister, R. Ounit, S. Lonardi, B. Morgenstern, rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLoS Comput. Biol.* **12**, e1005107 (2016).
40. S. Girotto, M. Comin, C. Pizzi, "Fast spaced seed hashing" in *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, R. Schwartz, K. Reinert, Eds. (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017), p. 7:1-7:14.
41. A. Markov, Extension of the limit theorems of probability theory to a sum of variables connected in a chain. *Dyn. Probab. Syst.* **1**, 552–577 (1971).
42. C. E. Shannon, A mathematical theory of communication. *Bell Syst. Tech. J.* **27**, 379–423 (1948).
43. R. Chiu, K. M. Nip, J. Chu, I. Birol, TAP: A targeted clinical genomics pipeline for detecting transcript variants using RNA-seq data. *BMC Med. Genomics* **11**, 79 (2018).
44. G Jacobson, *Succinct Static Data Structures*, (Carnegie Mellon University, Pittsburgh, PA, 1989).
45. S. Gog, M. Petri, Optimized succinct data structures for massive data. *Softw. Pract. Exper.* **44**, 1287–1314 (2013).
46. H. Mohamadi, J. Chu, B. P. Vandervalk, I. Birol, ntHash: Recursive nucleotide hashing. *Bioinformatics* **32**, 3492–3494 (2016).
47. J. S. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Softw.* **11**, 37–57 (1985).
48. O. J. Dunn, Estimation of the medians for dependent variables. *Ann. Math. Stat.* **30**, 192–197 (1959).
49. G. E. Alefeld, F. A. Potra, Y. Shi, Algorithm 748; enclosing zeros of continuous functions. *ACM Trans. Math. Softw.* **21**, 327–344 (1995).