JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 25, Number 5, 2018
© Mary Ann Liebert, Inc.
Pp. 467–479
DOI: 10.1089/cmb.2017.0258

# AllSome Sequence Bloom Trees

CHEN SUN,[1,*] ROBERT S. HARRIS,[2,*] RAYAN CHIKHI,[3] and PAUL MEDVEDEV[1,4,5]

## ABSTRACT

**The ubiquity of next-generation sequencing has transformed the size and nature of many databases, pushing the boundaries of current indexing and searching methods. One particular example is a database of 2652 human RNA-seq experiments uploaded to the Sequence Read Archive (SRA). Recently, Solomon and Kingsford proposed the Sequence Bloom Tree data structure and demonstrated how it can be used to accurately identify SRA samples that have a transcript of interest potentially expressed. In this article, we propose an improvement called the AllSome Sequence Bloom Tree. Results show that our new data structure significantly improves performance, reducing the tree construction time by 52.7% and query time by 39%–85%, with a price of upto 3×memory consumption during queries. Notably, it can query a batch of 198,074 queries in <8 hours (compared with around 2 days previously) and a whole set of *k*-mers from a sequencing experiment (about 27 million *k*-mers) in <11 minutes.**

**Keywords:** Sequence Bloom Trees, Bloom filters, RNA-seq, data structures, algorithms, bioinformatics.

## 1. INTRODUCTION

DATA STRUCTURES FOR INDEXING AND SEARCHING OF DATABASES have always been a core contribution of algorithmic bioinformatics to the analysis of biological data and are the building blocks of many popular tools (Mäkinen et al., 2015). Traditional databases may include reference genome assemblies, collections of known gene sequences, or reads from a single sequencing experiment. However, the ubiquity of next-generation sequencing has transformed the size and nature of many databases. Each sequencing experiment results in a collection of reads (gigabytes in size), typically deposited into a database such as the Sequence Read Archive (SRA) (Leinonen et al., 2011). There are thousands of experiments deposited into the SRA, creating a database of unprecedented size in genomics (four petabases, as of 2016). The SRA enables public access of the database through meta-data queries on the experiments' name, type, organism, etc. However, efficiently querying the raw read sequences of the database has remained out of reach for today's indexing and searching methods, until earlier this year (Solomon and Kingsford, 2016).

Departments of [1]Computer Science and Engineering and [2]Biology, Pennsylvania State University, University Park, Pennsylvania.
[3]CNRS, CRIStAL, University of Lille, Lille, France.
[4]Department of Biochemistry and Molecular Biology, Pennsylvania State University, University Park, Pennsylvania.
[5]Genome Sciences Institute of the Huck, Pennsylvania State University, University Park, Pennsylvania.

[*]These authors contributed equally to this work.

A preliminary version of this article appeared in RECOMB 2017.

Given a transcript of interest, an important problem is to identify all publicly available sequenced samples that express it. The SRA contains thousands of human RNA-seq experiments, providing a powerful database to answer this question. One approach is to use tools such as Trapnell et al. (2012), Patro et al. (2014), and Bray et al. (2016) to first identify transcripts present in each of the experiments; however, running these tools on a massive scale is time prohibitive [although cloud-enabled tools such as Rail-RNA (Nellore et al., 2016) are making inroads]. Moreover, they introduce biases and can easily miss a transcript that is supported by the reads. Another approach is to align the SRA reads to the transcript of interest; however, this approach is infeasible for such large data sets (Solomon and Kingsford, 2016).

Recently, Solomon and Kingsford (2016) proposed the Sequence Bloom Tree (SBT) data structure and demonstrated how it can accurately identify samples that may have the transcript of interest expressed in the read data. SBT was a breakthrough, allowing to query a set of 214,293 transcripts against a database of 2652 human RNA-seq experiments in just <4 days. The SBT is not intended to replace more thorough methods, like alignment, but is intended to be complementary, narrowing down the set of experiments for which a more rigorous investigation is needed.

In this article, we present the AllSome Sequence Bloom Tree (SBT-ALSO), a time and space improvement on the original SBT (denoted by SBT-SK). It combines three new ideas. The first one is a better construction algorithm based on clustering. The second one is a different representation of the internal nodes of the tree so as to allow earlier pruning and faster exploration of the search space. The final one is building a Bloom filter (BF) on the query itself. This allows quick execution of queries that are not just transcripts but are themselves large sequencing experiments.

We evaluate SBT-ALSO on the database of 2652 human RNA-seq runs used in Solomon and Kingsford (2016). SBT-ALSO reduces tree construction time by 52.7%, when given the BFs of the data sets. It reduces query time by 39%–85%, with a price of up to 3 × memory consumption. Notably, it can query a batch of 198,074 queries in <8 hours, compared with >2 days for SBT-SK. It can also query a whole set of *k*-mers from a sequencing experiment (about 27 million *k*-mers) in <11 minutes, compared with >23 hours by SBT-SK. Our software is open source and freely available by GitHub.[†]

## 2. RELATED WORK

This work falls into the general category of string pattern matching, where we are asked to locate all occurrences of a short pattern in a large text. In many cases, it is useful to preprocess the text to construct an index that will speed up future queries. The *k*-mer-index, trie, suffix tree, suffix array, Burrows–Wheeler transform, and FM-index are examples of such indices (Mäkinen et al., 2015). These form the basis of many read alignment tools such as BWA-MEM (Li, 2013) and Bowtie 2 (Langmead and Salzberg, 2012). Although many of these approaches are space and time efficient in their intended setting, they can nevertheless be infeasible on terabyte or petabyte scale data. Other approaches based on word-based indices (Navarro et al., 2000; Ziviani et al., 2000) and compressive genomics (Loh et al., 2012; Yu et al., 2015) do not help for the type of data and queries we consider in this article.

A BF is widely used to improve scalability by determining whether the pattern occurs in the text, without giving its location. It is a space-efficient data structure for representing sets that occasionally provides false-positive answers to membership queries (Bloom, 1970). For pattern matching, a BF can be constructed for all the constituent *k*-mers (strings of length of *k*) of the text. Then, if a high percentage of a pattern's constituent *k*-mers matches, the text is a potential match and a full search can be performed. BFs are used in several bioinformatics contexts such as assembly (Melsted and Pritchard, 2011; Chikhi and Rizk, 2013; Salikhov et al., 2013; Heo et al., 2014) to index and compress whole genome data sets (Rozov et al., 2014), and to compare sequencing experiments against whole genomes (Stranneheim et al., 2010).

When pattern matching against a database of read collections from sequencing experiments, additional factors need to be considered. First, the reads contain sequencing errors. Second, they only represent short fragments of the underlying DNA and are typically much shorter than the pattern. Third, there are many texts, each of which is its own sequencing experiment. The goal is to identify all texts that match the pattern. A simple way to adapt the BF idea to this case is to simply build a BF for every text and check the pattern

[†] SBT-ALSO GitHub repository. Available at: https://github.com/medvedevgroup/bloomtree-allsome

separately against every text's BF. A more sophisticated approach builds a tree to index the collection of BFs (Crainiceanu and Lemire, 2015). This Bloofi data structure was introduced in the context of distributed data provenance, but it was later adapted to the bioinformatics setting by Solomon and Kingsford (2016).

An orthogonal approach is the Bloom Filter Trie (BFT; Holley et al., 2015), which works similarly to a trie on the $k$-mers in all the texts. Each leaf contains a bitvector describing the texts in which that $k$-mer appears, and BFs are cleverly used inside the trie to ''jump down'' $\ell$ positions at a time, thus speeding up the trie traversal process. The BFT complexity scales up with the number of $k$-mers in the query, whereas SBT complexity scales up with the number of data sets. Thus the two approaches suggest orthogonal use cases. In particular, the BFT is very efficient for queries that are single $k$-mers, significantly outperforming the SBT. An approach that uses BFT to query longer patterns like those we consider in this article is promising but is not yet available.

There is also a body of work about storing and indexing assembled genomes (Ernst and Rahmann, 2013; Marcus et al., 2014; Baier et al., 2016; Liu et al., 2016; Minkin et al., 2016), which is part of the growing field of pangenomics (Computational Pan-Genomics Consortium, 2018). However, our work relates to the indexing of unassembled data (i.e., reads) as opposed to complete genomes. In addition to the topics specifically mentioned previously, there are other studies related to scaling up indexing methods (Marchet et al., 2016; Dolle et al., 2017), although the list here is in no way complete.

## 3. TECHNICAL BACKGROUND

### 3.1. Terminology

Let $x$ and $y$ be two bitvectors of the same length. The bitwise AND (i.e., intersection) between $x$ and $y$ is written as $x \cap y$, and the bitwise OR (i.e., union) is $x \cup y$. A bitvector can be viewed as a set of positions set to 1, and this notation is consistent with the notion of set union and intersection. The set difference of $x$ and $y$ is written as $x \backslash y$ and can be defined as $x \backslash y = x$ AND (NOT $y$). A BF is a bitvector of length $b$, together with $p$ hash functions, $h_1, \ldots, h_p$, where $b$ and $p$ are parameters. Each hash function maps a $k$-mer to an integer between 0 and $b-1$. The empty set is represented as an array of 0's. To add a $k$-mer $x$ to the set, we set the position $h_i(x)$ to 1, for all $i$. To check if a $k$-mer $x$ is in the set, we check that the position $h_i(x)$ is 1, for all $i$. In this article, we assume that the number of hash functions is 1 (see Section 6). Next, consider a rooted binary tree. The parent of a nonroot node $u$ is denoted as $parent(u)$, and the set of all the leaves of the subtree rooted at a node $v$ is denoted by $leaves(v)$. Let $lchild(u)$ and $rchild(u)$ refer to the left and right children of a nonleaf node $u$, respectively.

### 3.2. SBT

Let $Q$ be a nonempty set of $k$-mers, and let $B$ be a $k$-mer BF. Given $0 \leq \theta \leq 1$, we say that $Q$ $\theta$-matches $B$ if $|\{x \in Q \,:\, x \text{ exists in } B\}|/|Q| \geq \theta$. That is, the percentage of $k$-mers in $Q$ that are also in $B$ (including false-positive hits) is at least $\theta$. Solomon and Kingsford (2016) consider the following problem. We are given a database $D = \{D_1, \ldots, D_n\}$, where each $D_i$ is a BF of size $b$. The query is a $k$-mer set $Q$, and the result of the query should be the set $\{i \,:\, Q\ \theta-\text{matches } D_i\}$. The goal is to build a data structure that can construct an index on $D$ to support multiple future queries.

We make a distinction between the abstract data type that Solomon and Kingsford (2016) propose for the problem and their implementation of it. We call the first SBT, and the second SBT-SK [note that in Solomon and Kingsford (2016), no distinction is made and SBT refers to both]. A rooted binary tree is called a SBT of a database $D$ if there is a bijection between the leaf nodes and the elements of $D$. Define $B_\cup(u)$ for a leaf node $u$ as its associated database element and $B_\cup(u)$ for an internal node as $\bigcup_{i \in leaves(u)} B_\cup(i)$. Note that $B_\cup(u)$ of an internal node $u$ can be equivalently defined as $B_\cup(lchild(u)) \cup B_\cup(rchild(u))$. Each node $u$ then represents the set of database entries corresponding to the descendant leaves of $u$. In addition, the SBT provides an interface to construct the tree from a database, to query a $k$-mer set against the database, and to insert/delete a BF into/from the database. An example of an SBT is shown in Figure 2.

### 3.3. Sequence Bloom Tree-SK

We call the implementation of the SBT interface provided in Solomon and Kingsford (2016) as SBT-SK. In SBT-SK, each node $u$ is stored as a compressed version of $B_\cup(u)$. The compression is done using RRR

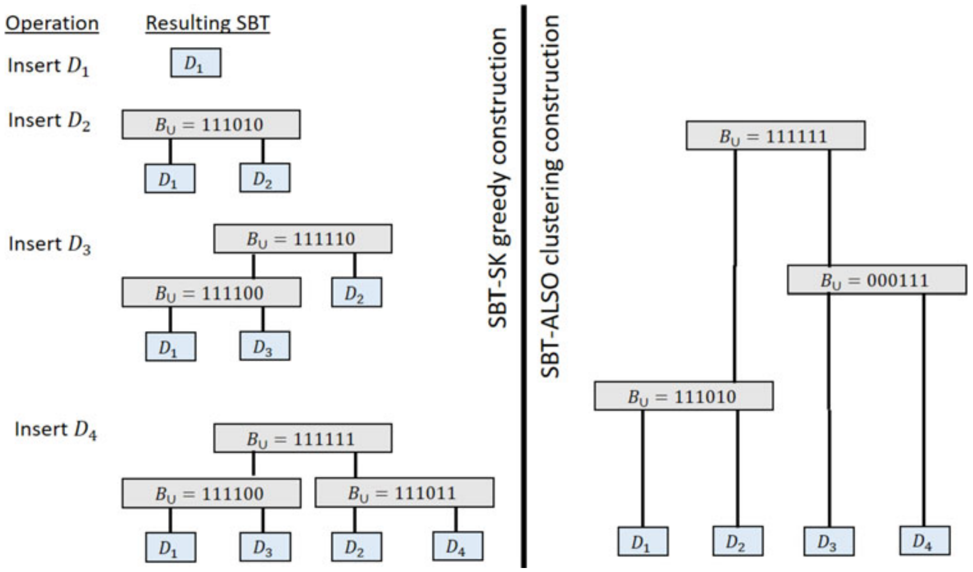

**FIG. 1.** Example of SBT-SK and SBT-ALSO construction algorithms for the database $D = \{D_1 = 111000, D_2 = 111010, D_3 = 000100, D_4 = 000011\}$. Leaves are shown in blue, internal nodes in gray. In this example, the data set can be partitioned into two types: 000xxx and 111xxx, based on the first three bits. In the SBT-SK construction, after the first two experiments are inserted (both of type 111xxx), they are destined to be in the two different sides of the tree (regardless of future insertions). Any future 111xxx type query will have to examine all the nodes. The SBT-ALSO construction, in contrast, groups together the experiments so that future 000xxx type or 111xxx type queries will have to examine only about half the nodes of the tree. ALSO, AllSome Sequence Bloom Tree; SBT, Sequence Bloom Tree.

(Raman et al., 2002) implemented in SDSL (Gog et al., 2014), which allows to efficiently test whether a bit is set to 1 without decompressing the bitvector. To *insert* a BF $B$ into an SBT $T$, SBT-SK does the following. If $T$ is empty, it just adds $B$ as the root. Otherwise, let $r$ be the root. If $r$ is a leaf, then add a new root $r'$ that is the parent of $B$ and $r$ and set $B_\cup(r') = B_\cup(r) \cup B$. Otherwise, take the child $v$ of $r$ that has the smallest Hamming distance to $B$, recursively insert $B$ in the subtree rooted at $v$, and update $B_\cup(r)$ to be $B_\cup(r) \cup B$. Note that because RRR-compressed bitvectors do not support bitwise operations, each bitvector must be first decompressed before bitwise operations are performed and then recompressed if any changes are made. The running time of an insertion is proportional to the depth of the SBT. To *construct* the SBT for a database, SBT-SK starts with an empty tree and inserts each element of the database one-by-one. Construction can take time proportional to $nd$, where $d$ is the depth of the constructed SBT. The left panel of Figure 1 provides an example of the construction algorithm. To *query* the database for a $k$-mer set $Q$, SBT-SK first checks whether $Q$ $\theta$-matches the root. If yes, then it recursively queries the children of the root. When the query hits a leaf node, it returns the leaf if $Q$ $\theta$-matches it.

Since SBT is designed to work on very large databases, its implementation should avoid loading the database into memory. In SBT-SK, each $B_\cup(u)$ is stored on disk and only loaded into memory when $u$ is being $\theta$-matched by a query. When there are multiple queries to be performed, SBT-SK will *batch* them together so that the $\theta$-matching of multiple queries to the same node will be performed simultaneously. Hence, each node needs to be loaded into memory only once per batch. We implement the same strategy in SBT-ALSO.

## 4. METHODS

We propose the AllSome SBT as an alternative implementation of the SBT abstract data type. In this section, we describe the construction and query algorithms. Insertion and deletion algorithms are the same as in SBT-SK, although some special care is needed. For completeness, they are described in the Appendix section.

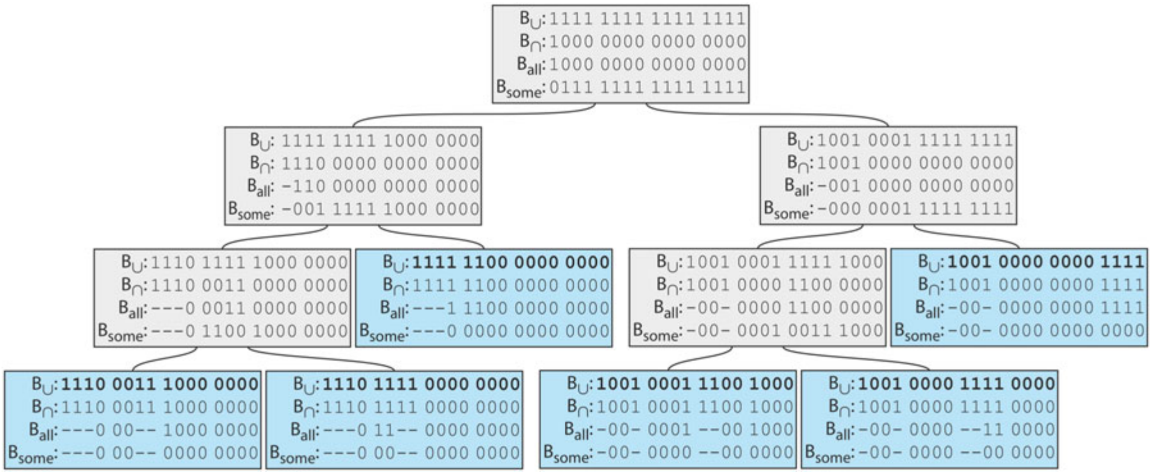

**FIG. 2.** Example SBT on $D = \{1110001110000000, 1110111100000000, 1111110000000000, 1001000111001000, 1001000011110000, 1001000000001111\}$. Leaves are shown in blue, internal nodes in gray. In SBT-ALSO, only $B_{all}$ and $B_{some}$ are explicitly stored, whereas in SBT-SK, only $B_{\cup}$ is stored. Bits present in $B_{all}$ at one node are shown as hyphens ('-') in the $B_{all}$ and $B_{some}$ of its descendants, but in the actual SBT-ALSO data structure, they are 0's.

### *4.1. AllSome node representation and regular query algorithm*

Define the intersection of leaves in the subtree rooted at a node $u$ as $B_{\cap}(u) = \bigcap_{i \in leaves(u)} B_{\cup}(i)$. Intuitively, we can partition the 1 bits of $B_{\cup}(u)$ into three sets: $B_{all}(u)$, $B_{some}(u)$, and $B_{\cap}(parent(u))$. $B_{all}(u)$ are the bits that appear in all of $leaves(u)$, excluding those in all of $leaves(parent(u))$. $B_{some}(u)$ are the bits in some of $leaves(u)$ but not in all. Both sets, therefore, exclude bits present in $B_{\cap}(parent(u))$. Formally, define

$$B_{all}(u) = B_{\cap}(u) \backslash B_{\cap}(parent(u))$$
$$B_{some}(u) = B_{\cup}(u) \backslash B_{\cap}(u).$$

At the root $r$, define $B_{\cap}(parent(r)) = \emptyset$. $B_{all}(u)$ and $B_{some}(u)$ are stored using two bitvectors of size $b$ compressed with RRR. $B_{\cup}(u)$ and $B_{\cap}(u)$ are not explicitly stored. We refer to this representation of the nodes using $B_{all}$ and $B_{some}$ as the AllSome representation (Fig. 2 gives an example).

When we receive a query $k$-mer set $Q$, we hash each $k$-mer to determine the list of BF bits corresponding to $Q$. These are a multiset of position indices (between 0 and $b-1$) stored as an array. We call these the list of *unresolved* bit positions. We also maintain two counters: the number of bit positions that have been determined to be 1 (*present*) and the number of bit positions determined to be 0 (*absent*). These counters are both initially 0. The query comparison then proceeds in a recursive manner. When comparing $Q$ against a node $u$, each unresolved bit position that is 1 in $B_{all}(u)$ is removed from the unresolved list and the present counter is incremented. Each unresolved bit position that is 0 in $B_{some}(u)$ is removed from the unresolved list and the absent counter is incremented. If the present counter is at least $\theta|Q|$, we add $leaves(u)$ to the list of $\theta$-matches and terminate the search of $u$'s subtree. If the absent counter exceeds $(1-\theta)|Q|$, we realize that $Q$ will not $\theta$-match any of the leaves in the subtree rooted at $u$ and terminate the search of $u$'s subtree. If neither of these holds, we recursively pass the two counters and the list of unresolved bits down to its children. When we reach a leaf, the unresolved list will become empty because $B_{some}$ is empty at a leaf, and the algorithm will necessarily terminate.

The idea behind the AllSome representation is that in a database of biologically associated samples, there are many $k$-mers that are shared between many data sets. In the SBT-SK representation, a query must continue checking for the presence of these $k$-mers at every node that it encounters. By storing at $u$ all the bits that are present in all the leaves of its subtree, we can count those bits as resolved much earlier in the query process—limiting the amount of bit look-ups performed. Moreover, we will often prune the search space earlier and decrease the number of bitvectors that need to be loaded from disk. A query that matches all the leaves of a subtree can often be resolved after just examining the root of that subtree. In the extreme case, the number of nodes examined in a search may be less than the number of database entries that are matched.

A second important point is that the size of the uncompressed bitvectors at each node is now twice as large as before. Because query time has a large I/O component, this has potential negative effects.

Fortunately, we observe that the compressed size of these bitvectors is roughly proportional to the number of 1's that are contained. By defining the AllSome representation as we do, the number of 1's in total in $B_{all}(u)$ and $B_{some}(u)$ is no more than the number of 1's in $B_{\cup}(u)$. Moreover, because we exclude $B_{\cap}(u)$ from all of $u$'s descendants, the number of 1's is less.

### 4.2. Construction algorithm

Except for large queries or large batches of queries, the running time of the query algorithm is dominated by the I/O of loading bitvectors into memory (Solomon and Kingsford, 2016). If the number of leaves that the query $\theta$-matches is localized within the same part of the SBT, then fewer internal nodes have to be explored and, hence, fewer bitvectors have to be loaded into memory. The SBT-SK construction algorithm is greedy and sensitive to the order in which the entries are inserted into the tree, which can lead to trees with poor localization (see Fig. 1).

To improve the localization property of the tree, we propose a nongreedy construction method based on agglomerative hierarchical clustering (de Hoon et al., 2004). Every $D_i$ is initially its own SBT, with its $B_{\cup}$ loaded into memory. At every step, two SBTs are chosen and joined together to form a new SBT. The new SBT has a root node $r$ with the left and right subtrees corresponding to the two SBTs being joined. $B_{\cup}(r)$ is computed as $B_{\cup}(lchild(r)) \cup B_{\cup}(rchild(r))$. To choose the pair of SBTs to be joined, we choose the two SBTs that have the smallest Hamming distance between the $B_{\cup}$ of their roots. The right panel of Figure 1 shows how our construction algorithm works.

Since each $B_{\cup}$ is a large bitvector, computing and maintaining the pairwise distances between all pairs are computationally expensive. Instead, we use the following heuristic. We fix a number $b' \ll b$ (e.g., $b' = 10^5 \ll 10^9 = b$) and then extract $b'$ bits from each $D_i$, starting from a fixed but arbitrary offset. We then run the mentioned clustering algorithm on this smaller database of extracted bitvectors.

The resulting topology is then extracted and used for constructing the $B_{all}$ and $B_{some}$ bitvectors for all the nodes. We process the nodes in a bottom-up manner. Initially, for all leaves $u$, we set $B_{all}(u) = B_{\cup}(u)$ and $B_{some}(u) = \emptyset$. For the general case, consider an internal node $u$ whose children $\ell$ and $r$ have already been processed. All bits that are set in both $B_{all}(l)$ and $B_{all}(r)$ go into $B_{all}(u)$:

$$B_{all}(u) = B_{all}(l) \cap B_{all}(r).$$

In addition, the $B_{all}$ bits of $\ell$ and $r$ must exclude those that are set in the parent $B_{all}(u)$. After computing $B_{all}(u)$, we can unset these bits:

$$B_{all}(v) = B_{all}(v) \backslash B_{all}(u), \ \ \text{where} v \in \{\ell, r\}. \tag{1}$$

Note that this is the only necessary update to the bitvectors of nodes in the subtree rooted in $\ell$ or $r$. Next, we must compute $B_{some}(u)$, which is the set of bits that exist in some of $u$'s children nodes but not all:

$$B_{some}(u) = B_{some}(\ell) \cup B_{some}(r) \cup B_{all}(\ell) \cup B_{all}(r).$$

Note that here we are using the $B_{all}$ after the application of Equation (1). This completes the necessary updates to the tree for a node $u$. These updates can be efficiently computed using bitwise operations on uncompressed bitvectors, so we keep them uncompressed in memory and only compress them when they are written to disk and are no longer needed. The total time for construction is proportional to $n$ and not to $nd$, as with SBT-SK. For completeness, we provide a more formal algebraic derivation of the update formulas in the Appendix section.

### 4.3. Large query algorithm

The "regular query" algorithm (Section 4.1) is designed with relatively small queries in mind (e.g., thousands of $k$-mers from a transcript). However, after performing a new sequencing experiment, it might be desirable to query the database for other similar samples. In such cases, the query would itself be a whole sequencing experiment, containing millions of $k$-mers. Our experimental results show that neither SBT-SK nor our own regular query algorithm is efficient for these large queries.

Although for small queries, the running time is dominated by the I/O of loading bitvectors into memory, for large queries, the time taken to look up the query $k$-mers in $B_{all}$ and $B_{some}$ of a node becomes the bottleneck. Let $B_Q$ be the BF of size $b$ for the $k$-mers in query $Q$. We propose an alternative "large query" algorithm that can be used whenever the number of $k$-mers in the query exceeds some predefined user threshold. This large query algorithm is identical to the regular one except in the way that the unresolved list is maintained and updated. The basic idea is that instead of checking each $k$-mer in $Q$ one-by-one, we can do bitwise comparisons using $B_Q$. Assume for the moment that there are no two $k$-mers in $Q$ that hash to the same position (recall that our BFs have only one hash function). In this case, the list of unresolved bit positions can be represented as the set of 1 positions in $B_Q$. At a node $u$, we first increment the *present* counter by the number of 1's in $B_Q \cap B_{all}(u)$ and update the unresolved bit positions to be $B'_Q = B_Q \backslash B_{all}(u)$. Then we increment the *absent* counter by the number of 1's in $B'_Q \backslash B_{some}(u)$ and update the unresolved bit positions to be $B''_Q = B'_Q \cap B_{some}(u)$. If the counters do not exceed their respective thresholds, then we pass them and the remaining unresolved bits ($B''_Q$) down to the children.

When there are $k$-mers that hash to the same bit positions, the mentioned algorithm can still be used as a heuristic. In fact, it can be shown that the hits returned by the mentioned heuristic algorithm are always a subset of the hits that are returned by an exact algorithm, since the heuristic's counter values are never greater than those of the exact algorithm. But, we can obtain an exact algorithm by modifying the mentioned heuristic to also maintain a list of bit positions that have multiple $k$-mers hashing to them. An entry of the list is a bit position and the number of $k$-mers that hash to it. Whenever we make a bitwise comparison involving $B_Q$, this list is scanned to convert number of bits to number of $k$-mers. When the list is small, this exact algorithm should not be significantly slower than the heuristic algorithm.

Unfortunately, computing bitwise operations cannot be efficiently done on RRR compressed bitvectors. To support the large query algorithm, the bitvectors are compressed using the Roaring (Chambi et al., 2015) scheme (abbreviated ROAR). ROAR bitmaps are compressed using a hybrid technique that allows them to efficiently support set operations on bitvectors (intersection, union, difference, etc.). However, we found that they generally do not compress as well as RRR on our data, leading to longer I/O times. In cases wherein both small and large queries are common, and query time is more important than disk space, both a ROAR and a RRR compressed tree can be maintained.

## 5. RESULTS

We implemented SBT-ALSO, building on the SBT-SK code base (Kingsford, 2016). Solomon and Kingsford (2016) already explored the advantages, disadvantages, and accuracy of the SBT approach as a way of finding experiments wherein the queried transcripts are expressed. Since SBT-ALSO gives identical query results as SBT-SK, we, therefore, focus our evaluation on its resource utilization. We used the same data set for evaluation as in Solomon and Kingsford (2016). This is the set of 2652 runs representing the entirety (at the time of Solomon and Kingsford, 2016) of human RNA-seq runs from blood, brain, and breast tissues at the SRA, excluding those sequenced with SOLID. In Solomon and Kingsford (2016), each sequencing run was converted to a $k$-mer BF ($b = 2 \cdot 10^9$, $k = 20$) by the Jellyfish $k$-mer-counting software (containing $k$-mers that occur greater than a file-dependent threshold, typically at least three occurrences). We downloaded these BFs from Kingsford (2016) and used them as our database. Per the results of Solomon and Kingsford (2016), this BF size leads to a false-positive rate of 0.5 for an individual BF. We performed experiments on an OpenStack instance with 12 vCPUs (Intel Xeon E312xx), 128 GB memory, and 4 TB network-mounted disk storage.

To choose the appropriate number of bits to use for clustering ($b'$), we randomly sampled 5000 bitvector pairs from the data set and computed their pairwise distances. We then computed distances for the same pairs using only $b'$ bits for various values of $b'$. The two distance metrics showed a high correlation ($r^2 = 0.9999874$) for $b' = 500,000$.

We then constructed SBT-SK and SBT-ALSO, as well as two other trees to help us separate out the contributions of the clustering algorithm from the AllSome representation. These two trees are SBT-SK+CLUST, which uses the $B_\cup$ node representation of SBT-SK but the SBT-ALSO clustering construction, and SBT-SK+AS, which uses the greedy construction of SBT-SK but the AllSome node representation of SBT-SK.

First, we compared the space and time used to construct SBT-SK and SBT-ALSO (Table 1). SBT-ALSO reduces the tree construction time by 52.7% and resulting disk space by 11.4%. It requires twice as much intermediate space, due to maintaining two uncompressed bitvectors for each node instead of just one.

To study the regular query performance, we downloaded all known transcripts at least $k$ bases long (198,074 of them) from Gencode (ver. 25). We then queried several subsets of transcripts against both trees,

Table 1. Construction Time and Space

| | *SBT-SK* | *SBT-ALSO* |
|---|---|---|
| Construction of tree topology (i.e., clustering) | N/A | 27 minutes |
| Construction of internal nodes | 56 hours 54 minutes | 26 hours 3 minutes |
| Peak memory usage | 726 MB | 908 MB |
| Temporary disk space | 1235 GB | 2469 GB |
| Final disk space | 200 GB | 177 GB |

Times shown are wall-clock times. A single thread was used. Note that the SBT-SK tree that was constructed for the purposes of this table differs from the tree used by Solomon and Kingsford (2016) and in our other experiments because the insertion order during construction was not the same as in Solomon and Kingsford (2016) (because it was not described there).

SBT-ALSO, AllSome Sequence Bloom Tree; N/A, not applied; SBT, Sequence Bloom Tree.

and measured the number of nodes examined for each query (Fig. 3) as well as the running time (Table 2). The results of all query experiments in this article were verified to be equivalent between the tested data structures. SBT-ALSO reduces the runtime by 39%–85%, depending on the size of the batch, likely due to the fact that the number of nodes examined per query is reduced by 52.7%, on average. Notably, SBT-ALSO was able to query a very large batch (198,074 queries) in <8 hours, whereas SBT-SK took >2 days. SBT-ALSO uses more memory than SBT-SK on larger batches.

To study the performance of the large query algorithm, we selected an arbitrary run from our database (SRR806782) and used Jellyfish (Marçais Kingsford, 2011) to extract all 20-mers that appear at least three times. These 27,546,676 *k*-mers formed one query. In heuristic mode, the large query algorithm was 22 times faster than the regular algorithm, but only detected 47 hits, which is a subset of the 50 hits by regular algorithm (Table 3). In the exact mode, the large query algorithm recovered all the hits (as expected) and was 18 times faster. Compared with SBT-SK, it was 155 times faster.

The clustering construction, even without the AllSome representation, significantly reduces the number of nodes that need to be examined per query (36.5% on average when comparing SBT-SK to SBT-SK+CLUST in Fig. 3). The improvement seems to be uniform regardless of the number of leaf hits. As expected, this leads to a significant improvement in the running time (19%–32%, Section 5).

The AllSome representation, without the clustering construction, also gives the benefit of allowing earlier query resolution, but the effect only becomes pronounced for queries that hit many leaves. For instance, queries that hit >800 leaves examined 27.4% less nodes in SBT-SK+AS then in SBT-SK. In the extreme case, there are 7 queries out of 1000, where the number of nodes examined is less than the number of leaf hits, something that is not possible with SBT-SK. However, the benefits of clustering construction and AllSome representation are synergistic: the multiplicative effect of their individual contributions (42.3% decrease in number of examined nodes) is less than the observed effect of their combined contributions (52.7%). In terms of the running time performance, the AllSome representation incurs the

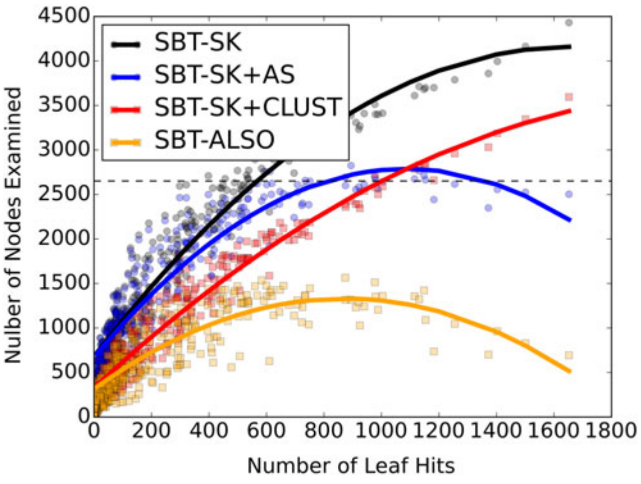

**FIG. 3.** Number of nodes examined per query for SBT-SK, SBT-ALSO, and two intermediate SBTs. A set of 1000 transcripts were chosen at random from Gencode set, and each transcript was queried against the four different trees. A dot represents a query and shows the number of matches in the database ($x$-axis) compared with the number of nodes that had to be loaded from disk and examined during the search ($y$-axis). For each tree (color), we interpolated a curve to show the pattern. The dashed horizontal line represents the hypothetical algorithm of simply checking whether the query $\theta$-matches against each of the database entries, one-by-one. For $\theta$, we used the default value in the SBT software ($\theta = 0.9$).

Table 2. Query Wall-Clock Run Times and Maximum Memory Usage for Batches of Different Sizes

| *No. of queries* | *SBT-SK* | *SBT-SK+AS* | *SBT-SK+CLUST* | *SBT-ALSO* |
|---|---|---|---|---|
| 1 | 1.2 min/301 MB | 2.7 min/301 MB | 0.9 min/299 MB | 0.5 min/301 MB |
| 10 | 4 min/305 MB | 8.3 min/319 MB | 3.3 min/304 MB | 2 min/313 MB |
| 100 | 7.7 min/315 MB | 13.7 min/346 MB | 6.5 min/317 MB | 4.7 min/353 MB |
| 1000 | 25.5 min/420 MB | 20.8 min/575 MB | 17.3 min/418 MB | 8.3 min/639 MB |
| 198,074 | 3082 min/22 GB | 1286 min/51 GB | 1910 min/23 GB | 463 min/63 GB |

For the batch of 1000 queries, we used the same 1000 queries as in Figure 3. For the batch of 100 queries, we generated three replicate sets, where each set contains 100 randomly sampled transcripts without replacement from the 1000 queries set. For the batch of 10 queries, we generated 10 replicate sets by partitioning 1 of the 100 query sets into 10 sets of 10 queries. For the batch of 1 query, we generated 50 replicate sets by sampling 50 random queries from Gencode set. The shown running times are the averages of these replicates. For $\theta$, we used the default value in the SBT software ($\theta=0.9$).

overhead of making two queries per active bit, instead of just one. This is more than compensated by a decrease in the number of active bits when the tree is clustered well. But, as the SBT-SK+AS column of Section 5 shows, the running time can actually deteriorate when the tree is not clustered.

## 6. DISCUSSION

In this article, we present an alternative implementation of the SBT that provides substantial improvements in query and construction time. We are especially effective for large batches of queries (6 times faster) or for large queries (155 times faster). Solomon and Kingsford (2016) make a convincing case that an efficient SBT implementation translates to an efficient and accurate solution to the broader problem of identifying RNA-seq samples that express a transcript of interest. They study the best parameter values of SBT ($\theta, k, b, p$) to achieve accuracy and speed for the broader problem. The focus of this article is on improving resource performance, and hence we do not revisit these questions; however, a more thorough exploration of the biological questions that the SBT can answer will be important moving forward.

The implications of using the SBT for queries that are themselves sequencing experiments were not explored in SBT-SK or here. The BFT (Holley et al., 2015), if adapted to multi-$k$-mer queries with $\theta$-matching, could prove to be powerful in this context. In general, the question of whether the percentage of matching $k$-mers is a good metric for comparing sequencing experiments is still open, and more investigation into how to best measure similarity is needed (see Murray et al., 2016). However, our large query algorithm opens the door for efficiently exploring the parameter space of $k$-mer-based approaches.

In contrast to SBT-SK, we do not currently support multiple hash functions. For the type of application considered in this article, Solomon and Kingsford (2016) demonstrated that one hash function is optimal. Yet, there may be other applications wherein multiple hash functions offer advantages. This may make SBT-ALSO, in its current state, less broadly applicable than SBT-SK. However, multiple hash functions could be implemented within the AllSome representation using partitioned BFs (where each hash function maps to a different bit array; Kirsch and Mitzenmacher, 2008). This remains a future work.

Table 3. Performance of Different Trees and Query Algorithms on a Large Query

| | *SBT-SK* | *SBT-ALSO* | | |
|---|---|---|---|---|
| | *Regular algorithm* | *Regular algorithm* | *Large exact algorithm* | *Large heuristic algorithm* |
| Query time | 1397 minutes 18 seconds | 195 minutes 33 seconds | 10 minutes 35 seconds | 8 minutes 32 seconds |
| Query memory | 2.3 GB | 4.7 GB | 1.3 GB | 1.2 GB |

We show the performance of SBT-SK and three query algorithms using SBT-ALSO compressed with ROAR: the regular algorithm, the large exact algorithm, and the large heuristic algorithm. We show the wall-clock run time and maximum RAM usage. We used $\theta=0.8$ for this experiment. The ROAR compressed tree was 190 GB (7.3% larger than the RRR tree).

ROAR, roaring.

Some of the ideas in this article were independently and concurrently discovered by Solomon and Kingsford (2017), appearing in the same issue.

# 7. APPENDIX

## 7.1. Insertion

If a tree is modified by the addition (or removal) of a leaf, the only nodes for which $B_\cup$ and $B_\cap$ can change are along the path from the leaf to the root. This fact, along with the definitions of $B_{all}$ and $B_{some}$, shows that it is sufficient to only consider changes in $B_{some}$ along that path, and in $B_{all}$ along that path and the siblings of those nodes.

To insert a new BF $B$, we follow the same strategy as SBT-SK. We insert $B$ starting at the root and recursively pass it down to the child $u$ that has the smallest Hamming distance between $B_\cup(u)$ and $B$. Although $B_\cup(u)$ is not explicitly stored in the SBT-ALSO, it can be recovered on the fly using the following equations:

$$B_\cap(u) = B_{all}(u) \cup B_\cap(parent(u)). \tag{2}$$

$$B_\cup(u) = B_{some}(u) \cup B_{all}(u) \cup B_\cap(parent(u)). \tag{3}$$

As we proceed down the tree, we must also update the appropriate bitvectors. Consider the insertion of $B$ into the subtree rooted at a node $u$. We inductively assume that the bitvectors of nodes outside the subtree rooted at $u$ have already been updated, that the bitvectors of nodes inside this subtree have been unchanged, and that $B_\cap(parent(u))$ is available in memory. We use the superscript *new* to denote the bitvectors of the nodes after $B$ is recursively passed down to one of the child's subtrees.

At $u$, observe that $B_\cup^{new}(u) = B_\cup(u) \cup B$ and $B_\cap^{new}(u) = B_\cap(u) \cap B$. This formula, together with $B_\cap(parent(u))$, is used to update $B_{all}(u)$ and $B_{some}(u)$, using their corresponding definitions. Assuming without loss of generality that $B$ will be passed down to the left child of $u$, the only other node that needs to be updated is the right child. Even though $B_\cup(rchild(u))$ and $B_\cap(rchild(u))$ remain unchanged, we need to update $B_{all}^{new}(rchild(u)) = B_{all}(rchild(u)) \backslash B$.

## 7.2. Deletion

Consider the deletion of an entry from the database. Let $v$ be the leaf representing the deleted entry, and let $v'$ be its sibling. We set $parent(v') = parent(parent(v'))$ and delete $v$ and $parent(v)$ from the tree, Next, we need to update the bitvectors of the tree.

Let $p$ be the path from the root down to $v'$. Let $p'$ be the nodes of $p$ along with the children of nodes in $p$. We use the superscript *new* to denote the bitvectors after the deletion, and omit the superscript to indicate bitvectors before the deletion. We will update the bitvectors in three passes. In the first pass, we will go down from the root to recover $B_\cap$ and $B_\cup$ for nodes in $p'$ and store them in active memory. In the second pass, we will go up from $v'$ and use the output of the first pass to calculate $B_\cup^{new}(u)$ and $B_\cap^{new}(u)$ for nodes in $p$. Note that $B_\cap^{new}(u) = B_\cap(u)$ and $B_\cup^{new}(u) = B_\cup(u)$ for nodes not in $p$. In the third pass, we will go up from $v'$ and use the output of the second pass to calculate $B_{all}^{new}(u)$ and $B_{some}^{new}(u)$ for all nodes on $p'$.

In the first pass, we can recover $B_\cap$ using Equations (2) and (3). In the second pass, we can compute (going up from the leaf)

$$B_\cup^{new}(u) = B_\cup^{new}(lchild(u)) \cup B_\cup^{new}(rchild(u)).$$

$$B_\cap^{new}(u) = B_\cap^{new}(lchild(u)) \cap B_\cap^{new}(rchild(u)).$$

In the third pass, we can compute

$$B_{all}^{new}(u) = B_\cap^{new}(u) \backslash B_\cap^{new}(parent(u)).$$

$$B_{some}^{new}(u) = B_\cup^{new}(u) \backslash B_\cap^{new}(u).$$

We note that with a smart implementation, the second and third passes can be combined, and the computation of $B_\cup(u)$ in the first pass can be done instead on the fly in the second pass. The mentioned algorithm also requires maintaining $O(d)$ bitvectors in memory, where $d$ is the depth of the tree. If memory is limited, then it is possible to read and write the bitvectors to disk for each node as it is being covered in a pass.

The running time of both an insertion and deletion is of the order of the depth of the tree. Performing an insertion/deletion requires performing bitwise operation on bitvectors, which can be done efficiently on a ROAR-compressed tree or an uncompressed tree. If RRR is being used, then, similar to SBT-SK, we need to uncompress nodes before processing them and recompress them after.

Finally, we note that if there are many modifications to the tree, the advantages of the initial clustering construction may dissipate. In this case, the tree can be reconstructed from scratch, incurring a time penalty but reducing the run time of future queries.

### *7.3. Formal derivation of update formulas for construction*

In Section 4.2, we presented the update rules for constructing $B_{all}$ and $B_{some}$ for the internal nodes of an SBT. Here, we give a formal derivation of the rules' correctness. First, let $DB(u)$ denote the database entries corresponding to the descendant leaves of a node $u$. Note that the subtree rooted at $u$ is, by definition, an SBT of $DB(u)$. At any point of the construction, we will have the invariant that if node $v$ was processed, then the subtree rooted at $v$ is a correct SBT-ALSO for $DB(v)$. For the base case, for all leaves $u$ we set $B_{all}(u) = B_{\cup}(u)$ and $B_{some}(u) = \emptyset$. For the general case, consider an internal node $u$ whose children have already been processed. We will use the superscript *new* to denote the values of the bitvectors for the new subtree rooted at $u$ and to distinguish it from those values passed up inductively from the trees of the children. An important point is that, for a child $v$, $B_{all}^{new}(v)$ may be different from $B_{all}(v)$. This is because once the SBT-ALSO of $DB(v)$ is incorporated into the SBT-ALSO of $DB(u)$, any bits that are set in $B_{all}^{new}(u)$ need to be unset in $B_{all}^{new}(v)$. Also, observe that for a root $r$ of an SBT-ALSO tree (e.g., $v$ in $DB(v)$ or $u$ in $DB(u)$), $B_{all}(r) = B_{\cap}(r)$ and $B_{\cup}(r) = B_{all}(r) \cup B_{some}(r)$. Applying our observations and definition, we can derive formula for $B_{all}^{new}(u)$, $B_{some}^{new}(u)$, and $B_{all}^{new}(v)$:

$$
\begin{aligned}
B_{all}^{new}(u) &= B_{\cap}^{new}(u) \\
&= B_{\cap}(lchild(u)) \cap B_{\cap}(rchild(u)) \\
&= B_{all}(lchild(u)) \cap B_{all}(rchild(u)).
\end{aligned}
$$

$$
\begin{aligned}
B_{all}^{new}(v) &= B_{\cap}^{new}(v) \backslash B_{\cap}^{new}(parent(v)) \\
&= B_{\cap}(v) \backslash B_{\cap}^{new}(u) \\
&= B_{all}(v) \backslash B_{all}^{new}(u).
\end{aligned}
$$

$$
\begin{aligned}
B_{some}^{new}(u) &= B_{\cup}^{new}(u) \backslash B_{\cap}^{new}(u) \\
&= \left( \bigcup_{w \text{ is a child of } u} B_{\cup}(w) \right) \backslash B_{all}^{new}(u) \\
&= \bigcup_{w} \big( B_{\cup}(w) \backslash B_{all}^{new}(u) \big) \\
&= \bigcup_{w} \big( (B_{all}(w) \cup B_{some}(w)) \backslash B_{all}^{new}(u) \big) \\
&= \bigcup_{w} \big( \big( B_{all}(w) \backslash B_{all}^{new}(u) \big) \cup \big( B_{some}(w) \backslash B_{all}^{new}(u) \big) \big) \\
&= \bigcup_{w} \big( \big( B_{all}(w) \backslash B_{all}^{new}(u) \big) \cup B_{some}(w) \big) \\
&= \bigcup_{w} \big( B_{all}^{new}(w) \cup B_{some}(w) \big).
\end{aligned}
$$

## ACKNOWLEDGMENT

This work was supported, in part, by NSF awards DBI-1356529, CCF-551439057, IIS-1453527, and IIS-1421908 to P.M.

## AUTHOR DISCLOSURE STATEMENT

No competing financial interests exist.

## REFERENCES

Baier, U., Beller, T., and Ohlebusch, E. 2016. Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform. *Bioinformatics.* 32, 497–504.

Bloom, B.H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM.* 13, 422–426.

Bray, N.L., Pimentel, H., Melsted, P., et al. 2016. Near-optimal probabilistic RNA-seq quantification. *Nat. Biotechnol.* 34, 525–527.

Chambi, S., Lemire, D., Kaser, O., et al. 2015. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exp.* 46, 709719.

Chikhi, R., and Rizk, G. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol.* 8, 1.

Computational Pan-Genomics Consortium. 2018. Computational pan-genomics: Status, promises and challenges. *Brief. Bioinform.* 19, 118–135.

Crainiceanu, A., and Lemire, D. 2015. Bloofi: Multidimensional Bloom filters. *Inf. Syst.* 54, 311–324.

de Hoon, M.J., Imoto, S., Nolan, J., et al. 2004. Open source clustering software. *Bioinformatics.* 20, 1453–1454.

Dolle, D.-D., Liu, Z., Cotten, M.L., et al. 2017. Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome Res.* 27, 300–309.

Ernst, C., and Rahmann, S. 2013. PanCake: A data structure for pangenomes, 35–45. *In German Conference on Bioinformatics, Volume 34*. Eds: Beibarth T., Kollmar, M., Leha, A., Morgenstern, B., Schultz, A.-K., Waack, S., and Wingender, E. Dagstuhl Publishing, Germany.

Gog, S., Beller, T., Moffat, A., et al. 2014. From theory to practice: Plug and play with succinct data structures, 326–337. *In International Symposium on Experimental Algorithms*. Gudmundsson, J., and Katajainen, J. Springer, Cham, Switzerland.

Heo, Y., Wu, X.-L., Chen, D., et al. 2014. BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics* 30, 1354–1362.

Holley, G., Wittler, R., and Stoye, J. 2015. Bloom Filter Trie—a data structure for pan-genome storage, 217–230. *In Algorithms in Bioinformatics*. Eds: Pop, M., and Touzet, M. Springer, London, UK.

Kingsford, C. 2016. SBT-SK software and data. Available at: www.cs.cmu.edu/%7Eckingsf/software/bloomtree. Accessed July 1, 2016.

Kirsch, A., and Mitzenmacher, M. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Struct. Algorithms.* 33, 187–218.

Langmead, B., and Salzberg, S.L. 2012. Fast gapped-read alignment with Bowtie 2. *Nat. Methods.* 9, 357–359.

Leinonen, R., Sugawara, H., and Shumway, M. 2011. The Sequence Read Archive. *Nucleic Acids Res.*39:D19–D21.

Li, H. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *Genomics.* arXiv preprint arXiv:1303.3997.

Liu, B., Zhu, D., and Wang, Y. 2016. deBWT: Parallel construction of Burrows–Wheeler transform for large collection of genomes with de Bruijn-branch encoding. *Bioinformatics.* 32, i174–i182.

Loh, P.-R., Baym, M., and Berger, B. 2012. Compressive genomics. *Nat. Biotechnol.* 30, 627–630.

Mäkinen, V., Belazzougui, D., Cunial, F., et al. 2015. *Genome-Scale Algorithm Design*. Cambridge University Press, Cambridge, UK.

Marçais, G., and Kingsford, C. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinformatics.* 27, 764–770.

Marchet, C., Limasset, A., Bittner, L., et al. 2016. A resource-frugal probabilistic dictionary and applications in (meta)genomics. *Data Structures and Algorithms.* arXiv preprint arXiv:1605.08319.

Marcus, S., Lee, H., and Schatz, M.C. 2014. SplitMEM: A graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics.* 30, 3476–3483.

Melsted, P., and Pritchard, J.K. 2011. Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics.* 12, 333.

Minkin, I., Pham, S., and Medvedev, P. 2016. TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics.* 33, btw609.

Murray, K.D., Webers, C., Ong, C.S., et al. 2016. kWIP: The k-mer weighted inner product, a de novo estimator of genetic similarity. *PLoS Comput. Biol.* 13, e1005727.

Navarro, G., De Moura, E.S., Neubert, M., et al. 2000. Adding compression to block addressing inverted indexes. *Inf. Retr.* 3, 49–77.

Nellore, A., Collado-Torres, L., Jaffe, A.E., et al. 2016. Rail-RNA: Scalable analysis of RNA-seq splicing and coverage. *Bioinformatics.* 33, 4033–4040.

Patro, R., Mount, S.M., and Kingsford, C. 2014. Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nat. Biotechnol.* 32, 462–464.

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

Rozov, R., Shamir, R., and Halperin, E. 2014. Fast lossless compression via cascading Bloom filters. *BMC Bioinformatics.* 15, 1.

Salikhov, K., Sacomoto, G., and Kucherov, G. 2013. Using cascading Bloom filters to improve the memory usage for de Brujin graphs, 364–376. *In* Darling, A., and Stoye, J., eds. *Algorithms in Bioinformatics, Volume 8126 of Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg.

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

Solomon, B., and Kingsford, C. 2017. Improved search of large transcriptomic sequencing databases using split Sequence Bloom Trees, 257–271. *In International Conference on Research in Computational Molecular Biology*. Ed: S. Cenk Sahinalp. Springer, Cham, Switzerland.

Stranneheim, H., Käller, M., Allander, T., et al. 2010. Classification of DNA sequences using Bloom filters. *Bioinformatics.* 26, 1595–1600.

Trapnell, C., Roberts, A., Goff, L., et al. 2012. Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks. *Nat. Protoc.* 7, 562–578.

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

Ziviani, N., de Moura, E.S., Navarro, G., et al. 2000. Compression: A key for next-generation text retrieval systems. *IEEE Computer.* 33, 37–44.

Address correspondence to:
*Prof. Paul Medvedev*
*Department of Computer Science and Engineering*
*The Pennsylvania State University*
*343J IST Building*
*University Park, PA 16802*

*E-mail:* pashadag@cse.psu.edu