# Improving Bloom filter performance on sequence data using $k$-mer Bloom filters

David Pellow[1⋆], Darya Filippova[2], and Carl Kingsford[2⋆⋆]

[1] The Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
[2] Computational Biology Department, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA

**Abstract.** Using a sequence's $k$-mer content rather than the full sequence directly has enabled significant performance improvements in several sequencing applications, such as metagenomic species identification, estimation of transcript abundances, and alignment-free comparison of sequencing data. Since $k$-mer sets often reach hundreds of millions of elements, traditional data structures are impractical for $k$-mer set storage, and Bloom filters and their variants are used instead. Bloom filters reduce the memory footprint required to store millions of $k$-mers while allowing for fast set containment queries, at the cost of a low false positive rate. We show that, because $k$-mers are derived from sequencing reads, the information about $k$-mer overlap in the original sequence can be used to reduce the false positive rate up to $30\times$ with little or no additional memory and with set containment queries that are only $1.3 - 1.6$ times slower. Alternatively, we can leverage $k$-mer overlap information to store $k$-mer sets in about half the space while maintaining the original false positive rate. We consider several variants of such $k$-mer Bloom filters ($k$BF), derive theoretical upper bounds for their false positive rate, and discuss their range of applications and limitations. We provide a reference implementation of $k$BF at https://github.com/Kingsford-Group/kbf/.

**Keywords:** Bloom filters, efficient data structures, $k$-mers

## 1. Introduction

Many algorithms central to biological sequence analysis rely, at their core, on $k$-mers—short substrings of equal length derived from the sequencing reads. For example, sequence assembly algorithms use $k$-mers as nodes in the de Bruijn graph [19, 10]; metagenomic sample diversity can be quantified by comparing the sample's $k$-mer content against a database [17]; $k$-mer content derived from RNA-seq reads can inform gene expression estimation procedures [9]; $k$-mer-based algorithms can dramatically improve compression of sequence [1, 11] and quality values [18] .

A single sequencing dataset could generate hundreds of millions of $k$-mers making $k$-mer storage a challenging problem. Bloom filters [2] are often used to store sets of $k$-mers since they require much less space than hash tables or vectors to represent the

⋆ Work performed at the Language Technologies Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA
⋆⋆ To whom correspondence should be addressed: carlk@cs.cmu.edu

same $k$-mer set while retaining the ability to quickly test for the presence of a specific $k$-mer at the cost of a low false positive rate (FPR) [4–8, 10–16]. For example, Bloom filters allow for efficient $k$-mer counting [8], can be used to represent de Buijn graphs in considerably less space [10], and can enable novel applications like inexact sequence search over very large collections of reads [14].

The small size of Bloom filters has allowed algorithms to efficiently process large amounts of sequencing data. However, smaller Bloom filter sizes have to be traded off against higher false positive rates: a smaller Bloom filter will incorrectly report the presence of a $k$-mer more often. Sequencing errors and natural variation noticeably increase $k$-mer set sizes, with recent long read data driving $k$-mer set sizes even higher due to this data's lower overall quality profiles. To support large $k$-mer sets, researchers can either increase the Bloom filter size, choose a more costly function to compute set containment, or attempt to reduce false positive rate through other means.

To eliminate the effects of Bloom filter's false positives when representing a probabilistic de Bruijn graph [10] — where two adjacent $k$-mers represent an implicit graph edge — one can precompute the false edges in the graph and store them separately [4]. The results of querying the Bloom filter for a $k$-mer are modified such that a positive answer is returned only if the $k$-mer is not in the critical false positive set. The size of the set of critical false positives is estimated to be $6Nf$ where $N$ is the number of nodes in the graph (and the number of $k$-mers inserted into the Bloom filter) and $f$ is the false positive rate of the original Bloom filter [12]. Cascading Bloom filters lower the FPR by storing a series of smaller nested Bloom filters that represent subsets of critical $k$-mers [12].

While specific Bloom filter applications achieved improved false positive performance by using additional data structures, these applications assume the FPR of general-purpose Bloom filters derived in the paper that presented them originally [2]. This FPR is calculated based on the assumption that elements inserted into the Bloom filter are independent. In biological sequencing applications which store $k$-mers, the elements are not independent: if all $k$-mers of a sequence are stored, then there is a $k-1$ character overlap between adjacent $k$-mers. The information about the presence of the $k$-mer's neighbours can be used to infer the $k$-mer itself is part of the set — without the use of additional storage.

We use $k$-mer non-independence to develop *$k$-mer Bloom filters* (*k*BF) with provably lower false positive rates. We first consider a *k*BF variant where we are able to achieve more than threefold decrease in FPR with no increase in required storage and only a modest delay in set containment queries ($1.2$–$1.3\times$ slower when compared to classic Bloom filter). We then consider a *k*BF with a stricter set containment criteria which results in more than 30-fold decrease in FPR with a modest increase in required storage and up to $1.9\times$ delay in set containment queries.

Since the existence of $k$-mers in the Bloom filter can be inferred from the presence of neighbouring $k$-mers, we can also drop certain $k$-mers entirely, sparsifying the Bloom filter input set. We implement sparsifying *k*BFs and achieve $k$-mer sets that are $51$–$60\%$ the size of the original set with a slightly lower FPR at the cost of slower set containment queries.

The space and speed requirements vary between diffent *k*BF variants allowing for a multitude of applications. In memory-critical algorithms, such as sequence assembly [10] and search [14], sparse *k*BF can lower memory requirements allowing to process larger read collections. Applications relying on $k$-mers for error correction [15] or classification [17] may benefit from using *k*BF with guaranteed lower false positive rates to confidently identify sequencing errors and to distinguish between related organisms in the same clade.

## 2. Reducing false positive rate using neighbouring $k$-mers

When testing a Bloom filter for the presence of the query $k$-mer $q$, for example `AATCCCT` (see Figure 1), the Bloom filter will return a positive answer — which could be a true or a false positive. However, if we query for the presence of neighbouring $k$-mers `xAATCCC` $k$-mers (where $x$ is one of $\{A, C, G, T\}$) and receive at least one positive answer, we could be more confident that `AATCCCT` was indeed present in the Bloom filter. There is a non-zero chance that $q$ is a false positive and its neighbour is itself a true positive, however, this is less likely than the chances of $q$ being a false positive and thus lowers the false positive rate. We formally introduce the *k*BF and derive the probabilities of such false positive events below.

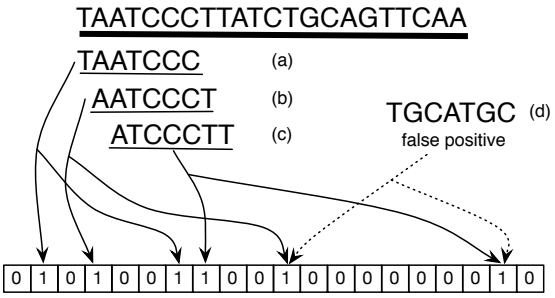


**Fig. 1.** The $k$-mers from a sequence are stored in a Bloom filter. False positives could occur when the bits corresponding to a random $k$-mer not in the sequence are set because of other $k$-mers which are in the Bloom filter. The true $k$-mers from the sequence all share sequence overlaps with other true $k$-mers from the sequence. We show how this overlap can be used to reduce the false positive rate and sparsify the set of $k$-mers stored in the *k*BF.

### 2.1 One-sided $k$-mer Bloom filter

We define a *k*BF that only checks for the presence of a single overlapping neighbour when answering a set containment query as a *one-sided kBF* (1-*k*BF). Each $k$-mer $q$ observed in the sequence or collection of sequencing reads is inserted into a Bloom filter $B$ independently in the standard way. To test for $q$'s membership in a 1-*k*BF, the

Bloom filter $B$ is first queried for $q$. If the query is successful, then $q$ is either in the true set of $k$-mers $U$, or is a false positive. If $q \in U$ and all $k$-mers in $U$ were added to the Bloom filter, the set containment query for $q$'s neighbour should return "true". We generate all eight potential left and right neighbours for $q$ and test whether $B$ returns true for any of them (see Alg. 1). Under the assumption that every read or sequence is longer than $k$, every $k$-mer will have at least one neighbour in the right or left direction.

**Algorithm 1** One-sided *k*BF *contains* functions

```
function ONE-SIDED_KBF_CONTAINS(query)
    if BF.CONTAINS(query) then
        return CONTAINS_SET(NEIGHBOUR_SET(query))
    return false

function CONTAINS_SET(set)
    for kmer ∈ set do
        if BF.CONTAINS(kmer) then return true
    return false
```

### 2.2 Theoretical FPR for a one-sided $k$-mer Bloom filter

We show that the theoretical upper bound for the FPR of a one-sided *k*BF is lower than that for the classic Bloom filter [2]. Suppose we inserted $n$ unique $k$-mers into a Bloom filter of length $m$ using $h$ hash functions. Then the expected proportion of bits that are still 0 is $E = (1 - 1/m)^{hn}$ and the actual proportion of zeros, $p$, is concentrated around this mean with high probability [3]. The false positive rate $f$ is:

$$f = (1-p)^h \approx (1-E)^h = \left(1 - (1 - 1/m)^{hn}\right)^h . \tag{1}$$

Let $q'$ be $q$'s neighbour that overlaps $q$ by $k-1$ characters on either the left or the right side, and let $t_k$ be a probability that a random $k$-mer is a true positive (i.e. is present in the set $U$). We assume further that events such as "$q$ is a false positive" and "$q'$ is a false positive" are independent since the false positives result from bits being set by uniform random hashes of other $k$-mers inserted into the Bloom filter. Then the chances that we get a false positive when testing for the presence of a random $k$-mer $q$ and one of its eight neighbours $q'$ are:

$$\begin{aligned}
f' &= f \cdot \Pr(\text{BF returns "True" for at least one of the adjacent } k\text{-mers}) && (2)\\
&= f \cdot (1 - \Pr(\text{BF returns "False" for every adjacent } k\text{-mer})) && (3)\\
&= f \cdot \left(1 - \Pr(\text{BF returns "False" for an adjacent } k\text{-mer})^8\right) && (4)\\
&= f \cdot \left(1 - (1 - \Pr(\text{BF returns "True" for an adjacent } k\text{-mer}))^8\right) && (5)\\
&= f \cdot \left(1 - (1 - (f + t_k))^8\right) . && (6)
\end{aligned}$$

Assuming that $k$-mers are uniformly distributed, we can estimate $t_k$ as the chance of drawing a $k$-mer from the set $U$ giving the set of all possible $k$-mers of length $k$, or $t_k = |U|/4^k$. For reasonably large values of $k \geq 20$, $t_k$ will be much smaller than $f$ allowing us to estimate an upper bound on $f'$:

$$f' < f \cdot \left(1 - (1 - 2f)^8\right) \tag{7}$$

quantifying how much lower $f'$ is than the false positive rate $f$ for the classic Bloom filter.

### 2.3 Two-sided $k$-mer Bloom filter

The FPR could be lowered even more by requiring that there is a neighbouring $k$-mer extending the query $k$-mer in both directions in order to return a positive result. This is a two-sided $k$-mer Bloom filter, 2-*k*BF. However, this requires dealing with $k$-mers at the boundary of the input string. In Figure 1, it can be seen that the first $k$-mer (`TAATCCC`) only has a right neighbour and no left neighbouring $k$-mers. We call this an *edge $k$-mer,* which must be handled specially, otherwise the 2-*k*BF would return a false negative.

To avoid this, 2-*k*BF maintains a separate list that contains these edge $k$-mers. We augment the Bloom filter with a hashtable EDGE_k-mer_set that stores the first and last $k$-mers of every sequence in order to handle edge cases. When constructing the *k*BF from a set of sequence reads, the first and last $k$-mer of each read are stored. Since reads can overlap, many of the read edge $k$-mers, will not be true edges of the sequence. After all the reads have been inserted into the Bloom filter, each of the stored $k$-mers is checked to see if it is an edge $k$-mer of the sequence and if it is, then it is saved in the final table of edge $k$-mers. The only $k$-mers that will be stored in the final edge table are those at the beginning and end of the sequence, or those adjacent to regions of zero coverage. Pseudocode for querying a two-sided $k$-mer Bloom filter is given in Alg. 2.

**Algorithm 2** two-sided *k*BF *contains* function

```
1:  function TWO-SIDED_KBF_CONTAINS(query)
2:      if BF.CONTAINS(query) then
3:          Contains_left ← CONTAINS_SET(LEFT_NEIGHBOUR_SET(query))
4:          Contains_right ← CONTAINS_SET(RIGHT_NEIGHBOUR_SET(query))
5:          if Contains_right == true and Contains_left == true then
6:              return true
7:          if Contains_right == true or Contains_left == true then
8:              if EDGE_k-mer_SET.contains(query) then
9:                  return true
10:     return FALSE
```

### 2.4 Theoretical FPR for a two-sided $k$-mer Bloom filter

Ignoring edge $k$-mers for simplicity and following the same assumptions and derivation as in Sec 2.2, the FPR for two-sided $k$BF, $f'$, is

$$f' = f \cdot \Pr(\text{BF returns "True" for at least one of the left adjacent } k\text{-mers}) \cdot \Pr(\text{BF returns "True" at least one of the right adjacent } k\text{-mers}). \tag{8}$$

This leads to

$$f' = f \cdot \left(1 - (1 - (f + t_k))^4\right)^2. \tag{9}$$

An upper bound for this expression can be estimated as:

$$f' < f \cdot \left(1 - (1 - 2f)^4\right)^2. \tag{10}$$

## 3. Using sequence overlaps to sparsify $k$-mer sets

We can use the assumption that the set of $k$-mers to be stored, $U$, contains $k$-mers derived from an underlying string $T$ to reduce the number of $k$-mers that must be stored in $B$ without compromising the false positive rate. If we want to store a set $U$, we can choose a subset $K \subseteq U$ that will be stored in $B$. The idea is that every $k$-mer $u \in U$ will have some neighbours that precede it and some that follow it in the string $T$.

Let $L_u \subset U$ be a set of $k$-mers that occur before $u$ in $T$, and let $R_u \subset U$ be a set of $k$-mers that occur after $u$ in $T$. If we can guarantee that there is at least one $k$-mer of $L_u$ and at least one $k$-mer of $R_u$ that are close to $u$ stored in $B$, then we can infer the presence of $u$ from the presence of $v \in L_u$ and $w \in R_u$ without having to store $u \in B$. By reducing the $k$-mers that must be kept in $B$, we can maintain the set $U$ using a smaller filter $B$. For example, in Figure 1 the $k$-mer `AATCCCT` is preceded by `TAATCCC` and followed by `ATCCCTT`. If these two $k$-mers are stored in $B$ then the presence of the middle $k$-mer `AATCCCT` in the sequence can be inferred without having to store it in the Bloom filter.

More formally, define $P_{vu}$ to be the set of positions of $k$-mer $v$ occurring before $u$ in $T$, and let $A_{uw}$ be the set of positions of $k$-mer $w$ occurring after $u$ in $T$. We then define, for $v \in L_u$ and $w \in R_u$, the set of all distances between occurrences of these $k$-mers that span $u$:

$$S_u(v, w) = \{i_w - i_v \mid i_v \in P_{vu}, i_w \in A_{uw}\}.$$

For some *skip length* $s$, if we can guarantee that $\min S_u(v, w) \leq s$ for some $v, w \in K$ then we can infer the presence of $u$ without storing it in the Bloom filter by searching for neighbouring $k$-mers that satisfy $\min S_u(v, w) \leq s$. This leads to the following $k$-mer sparsification problem:

**Problem 1** *Given a set of $k$-mers $U$, find a small subset $K \subset U$ such that for all $u \in U$, either $u \in K$ or there is a $k$-mer $v \in K \cap L_u$ and $w \in K \cap R_u$ with $\min S_u(v, w) \leq s$.*

We call this problem the *relaxed* $k$-mer sparsification problem. When we require exactly $s$ skipped $k$-mers between those $k$-mers chosen for $K$ we have the *strict* $k$-mer sparsification problem:

**Problem 2** *Given a set of $k$-mers $U$, find a small subset $K \subset U$ such that for all $u \in U$, either $u \in K$ or there is a $k$-mer $v \in K \cap L_u$ and $w \in K \cap R_u$ with $s \in S_u(v, w)$.*

These sparsification problems would be easy if we could observe $T$ — a solution would be to select every $s$th $k$-mer (Approach 3 below). However, we assume that we see only short reads from $T$, and must select $K$ as best as possible. Below, we propose three solutions to the $k$-mer sparsification problem that are appropriate in different settings.

*Approach 1: Best index match per read sparsification (kmers come from reads; arbitrary $s$).* $K$ will be built greedily by choosing $k$-mers from each read. Given a read $r$, we choose every $s$th $k$-mer starting from a particular index $i_r$, choosing $i_r$ such that the set of $k$-mers $K_r$ chosen for this read has the largest intersection with the set of $k$-mers $K$ chosen so far.

*Approach 2: Sparsification via approximate hitting set ($k$-mers come from reads; $s = 1$).* When $s = 1$, the relaxed $k$-mer set sparsification problem can also be formulated as a minimal hitting set problem: For each $k$-mer $k \in U$, create a set $L_k$ which includes $k$ and every $k$-mer which immediately preceded it in some read, and a set $R_k$ which includes $k$ and every $k$-mer which immediately followed it in some read. Let $L = \{L_k : k \in U\}$ and $R = \{R_k : k \in U\}$. A solution to the minimal hitting problem choses a minimally sized set $K$ such that at least one $k$-mer from $K$ is in every set in $R$ and $L$: $\forall N \in \{R \cup L\}\ \exists k \in K$ s.t. $k \in N$ and $|K|$ is minimized. We use a greedy approximation for the hitting set problem to choose $K$, the sparse set of $k$-mers. In each step of the greedy approximation, we add the $k$-mer which hits the most sets in $L \cup R$ to $K$.

*Approach 3: Single sequence sparsification (kmers come from a known sequence $T$; arbitrary $s$).* In the special case where input sequences are non-overlapping (e.g. a genome or exome) rather than multiple overlapping sequences (e.g. the results of a sequencing experiment), we solve the strict $k$-mer sparsification problem by taking each $k$-mer starting from the beginning of the sequence and then skipping $s$ $k$-mers. This is a simple and fast way to choose the sparse set $K$, but restricted only to this special case, and will not work for sparsifying the $k$-mers from a set of reads generated in a sequencing experiment. It is useful, for example, if the input sequences are a reference genome which will be queried against.

Once $K$ has been chosen, a sparse *k*BF can be queried for $k$-mers from $U$ using the pseudocode in Alg. 3. Two different query functions are given: RELAXED-CONTAINS for when $K$ satisfies the conditions of Problem 1, and STRICT-CONTAINS for when $K$ satisfied the conditions of Problem 2. We call two $k$-mers with $s$ skipped $k$-mers between them $s$-distant neighbours. The helper functions CONTAINS_NEIGHBOURS determine whether $k$-mers neighbouring the query $k$-mer at specified distances to the left and right are present and DECIDE_PRESENT determines whether the query is present depending on whether it has neighbouring $k$-mers or is an edge. Note that the sparse *k*BF also maintains a set of edge $k$-mers which is queried when a $k$-mer has neighbours in one direction but not the other.

**Algorithm 3** sparse *k*BF *contains* functions

**function** DECIDE_PRESENT($query, Contains\_left, Contains\_right$)
    **if** Contains_right == **true and** Contains_left == **true then**
        **return true**
    **if** Contains_right == **true or** Contains_left == **true then**
        **if** EDGE_$k$-mer_SET.contains($query$) **then**
            **return true**
    **return false**

**function** STRICT-CONTAINS_NEIGHBOURS($query, left\_dist, right\_dist$)
    Contains_left ← CONTAINS_SET( S_DISTANT_LEFT_NEIGHBOUR_SET($query, left\_dist$))
    Contains_right ← CONTAINS_SET(S_DISTANT_RIGHT_NEIGHBOUR_SET($query, right\_dist$))
    **return** DECIDE_PRESENT($query$, $Contains\_left$, $Contains\_right$)

**function** RELAXED-CONTAINS_NEIGHBOURS($query, l\_dist, r\_dist$)
    Contains_left ← CONTAINS_SET( $\bigcup_{i \leq l\_dist}$ S_DISTANT_LEFT_NEIGHBOUR_SET($query, i$))
    Contains_right ← CONTAINS_SET( $\bigcup_{i \leq r\_dist}$ S_DISTANT_RIGHT_NEIGHBOUR_SET($query, i$))
    **return** DECIDE_PRESENT($query$, $Contains\_left$, $Contains\_right$)

**function** STRICT-CONTAINS($query, s$)
    **if** BF.CONTAINS($query$) **then**
        **if** STRICT-CONTAINS_NEIGHBOURS($query, s, s$) **then**
            **return true**
    **for** $i \leftarrow 0$ **to** $s - 1$ **do**
        **if** STRICT-CONTAINS_NEIGHBOURS($query, i, s - (i + 1)$) **then**
            **return true**
    **return false**

**function** RELAXED-CONTAINS($query, s$)
    **if** BF.CONTAINS($query$) **then**
        **if** RELAXED-CONTAINS_NEIGHBOURS($query, s, s$) **then**
            **return true**
    **else**
        **for** $i \leftarrow 0$ **to** $s - 1$ **do**
            **if** RELAXED-CONTAINS_NEIGHBOURS($query, i, s - (i + 1)$) **then**
                **return true**
    **return false**

## 4. Results and Discussion

We test the performance of the proposed *k*BFs on a variety of sequencing experiments and compare to classic Bloom filters. For each test, we store the $k$-mers from the input file in the *k*BF and create a query set by mutating $k$-mers from the input. We test on multiple species and types of experiments that could typically be used in applications that require Bloom filters over a range of input file sizes. The different data sets are summarized in Table 1.

| Accession | Type | Read count | Read length |
|---|---|---|---|
| ERR233214_1 | WGS of *P. aeruginosa* | 7,571,879 | 92 |
| SRR1031159_1 | metagenomic, WGS | 674,989 | 101 |
| SRR514250_1 | metagenomic, WGS | 44,758,957 | 100 |
| SRR553460 | human RNA-seq | 66,396,200 | 49 |
| chr15 | human chromosome (hg19) | 1 | 81Mbp |

**Table 1.** Read sets on which *k*BF variants were tested. Only reads without "N" bases were included.

For all tests we used a $k$-mer length of $k = 20$ and 2 hash functions in the underlying Bloom filter. This choice of $k$ is long enough that only a fraction of all possible $k$-mers are present in reasonably large datasets and shorter than all read lengths. The Bloom filter length is 10 times the number of $k$-mers inserted into it for each of the input files. For one-sided *k*BF and two-sided *k*BF, the underlying Bloom filter will be exactly the same as the classic Bloom filter they are compared to. For sparse *k*BF, the smaller sparse $k$-mer set is stored, so the underlying Bloom filter is smaller. The sparse *k*BFs use a skip length of $s = 1$. The implementations of the *k*BF variants described wrap around the basic Bloom filter implementation from libbf (http://mavam.github.io/libbf), which is used for the classic Bloom filter.

To create a query $k$-mer set for testing we randomly select (uniformly, with replacement) 1 million $k$-mers from the input file and mutate one random base. This creates a set of $k$-mers that are close to the real set, and will therefore have realistic nucleotide sequences while still providing many negative queries to test the false positive rate. For one experiment (SRR1031159) we also query with 1 million true queries (not mutated) to determine the worst-case impact on query time.

### 4.1 One-sided and two-sided *k*BF performance

The one-sided and two-sided *k*BF implementations achieve substantially better FPRs than the classic Bloom filter (Table 2) at the cost of some query time overhead (Figure 2). For the one million mutated queries, only about one quarter of the queries are true positives, and one-sided *k*BF and two-sided *k*BF take 1.3 and 1.6 times as long to perform the queries, respectively. In the worst case, when all of the queries are true positives (SRR1031159 TP) one-sided *k*BF and two-sided *k*BF are 3.3 and 5.8 times slower respectively, while the speed of the classic Bloom filter does not change.

One-sided *k*BF has a FPR less than one third of the classic Bloom filter FPR at a cost of an extra one third the query time. Query times are extremely low, and this extra cost totals less than half a second to perform one million.

The two-sided *k*BF requires a special data structure which stores the set of edge $k$-mers that may not be found because there is no adjacent $k$-mer on one side. The total number of $k$-mers and the number of $k$-mers in the edge set of each file are compared in Table 3. There is also extra memory and speed overhead during the two-sided *k*BF creation: as the sequence file is read in and split into $k$-mers, a set of $k$-mers at the edges of all reads (which could potentially be sequence edges that need to be stored

| Accession | Classic | 1-$k$BF | 2-$k$BF | sparse best match | sparse hitting set |
|---|---|---|---|---|---|
| ERR233214_1 | 0.0329 | 0.0104 | 0.0009 | 0.0284 | 0.0311 |
| SRR1031159_1 | 0.0329 | 0.0104 | 0.0009 | 0.0279 | 0.0306 |
| SRR514250_1 | 0.0329 | 0.0106 | 0.0010 | 0.0290 | - |
| SRR553460 | 0.0329 | 0.0104 | 0.0009 | 0.0285 | 0.0314 |
| Chr15 | 0.0328 | 0.0104 | 0.0009 | 0.0284 | 0.0309 |
| Theoretical FPR: | $0.0328 <$ | $0.0138 <$ | 0.0019 | - | - |

**Table 2. False positive rates.** Comparison of FPRs for classic Bloom filters, and the different $k$BF implementations. The theoretical FPRs are also shown in the last row (calculated according to Eqns. 1, 7, and 10). Hitting set sparsification uses the relaxed CONTAINS function, while best match uses the strict CONTAINS function. The sparse hitting set results for SRR514250_1 are missing since the method never completed on this data set.

separately) is maintained. After $k$-mers are inserted into the Bloom filter, the edges are checked, and only the true sequence edges, which do not have neighbouring $k$-mers on both sides, are stored. We do not optimize the $k$BF implementations for the one time cost of creating the $k$-mer set and populating the Bloom filter but note that the potential edge set could be pruned on the fly, keeping it smaller than the maximum size achieved here. We report the number of potential edge $k$-mers stored in the edge set, and the amount of extra time to check the edges in Table 3. In all tested cases, the number of edge $k$-mers stored is a small fraction of the total number of $k$-mers, reaching at most 6% of the total. It is smallest when there are very few true sequence edges (in the single chromosome) and can be large if there are many reads with errors in the edge $k$-mers or many areas with zero coverage. If this overhead can be tolerated, applications could use two-sided $k$BF to achieve significantly lower FPRs.

Two-sided $k$BF provides a FPR that is 30$\times$ smaller than classic Bloom filters with a small query time penalty. Two-sided $k$BF also has a one-time cost of initialization to keep track of all potential edge $k$-mers and then determine the true edges. The extra time for this is only a small fraction of the total initialization cost. However, a large number of potential edge $k$-mers are stored during initialization. This number depends on the number of unique reads, and for the data sets with few long reads, i.e. the single chromosome, there is very little overhead, while when there are many reads, the first and last $k$-mer of each read could be stored.

### 4.2 Sparse *k*BF performance

The sparse $k$BF implementations achieve slightly better FPRs than the classic Bloom filters while using a smaller filter. We report the FPRs for the best index match and hitting set implementations of sparse $k$BF in Table 2. We do not report specific results for single sequence sparsification (Approach 3) since we found in practice the results are the same as for best match sparsification in the cases where it is relevant. When a sparse set of $k$-mers is used, sparse $k$BF is able to use the sequence overlap to recover a similar FPR for this smaller set of $k$-mers.

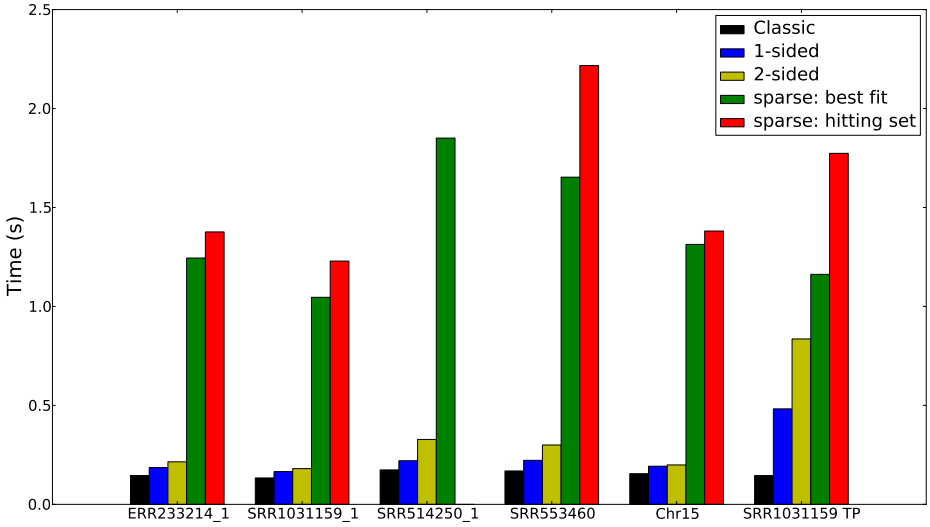


**Fig. 2. Query times.** Comparison of the time to query 1 million $k$-mers in classic BF and the different *k*BF implementations (average of 10 runs)

The sparsification performance of the different implementations is compared in Table 4. The sparsification methods perform well, with the best match achieving close to the ideal size of one half the number of $k$-mers. The hitting set sparsification method does not perform as well, choosing a $k$-mer set that is roughly 10% larger than the best match method.

Sparse *k*BF queries are significantly slower than classic Bloom filter queries. The speeds to perform 1 million queries for the classic BF and the different sparse *k*BF implementations are shown in Figure 2. The time overhead of querying neighbouring $k$-mers is about ten times that for a classic Bloom filter, but is still only around 1-2 seconds for one million queries. In memory-constrained applications, it could be worth paying this timing penalty for smaller $k$-mer sets. The time overhead will grow exponentially as $s$ is increased, but even very small $s$ (such as $s = 1$ shown in our experiments) significantly reduces the size of the stored $k$-mer set. Similarly, as $s$ increases, the FPR will increase, but as we showed here, for small $s$, the FPR is comparable to the FPR of a classic Bloom filter.

The hitting set sparsification implementation has a very large memory footprint and takes a lot of time to choose the sparse $k$-mer set. For the largest file (SRR514250), the implementation uses up all available RAM and does not complete even after running for several days. Table 5 compares the total time to split the input sequences into $k$-mers, choose the $k$-mer set, determine the edge $k$-mers and populate the Bloom filter for the different sparsification implementations and two-sided *k*BF. We compare with two-sided *k*BF because it also has edge $k$-mers, making it the most similar non-sparse implementation to the sparse *k*BF implementations. The memory overhead of initial-

| Accession | # of $k$-mers | # of edge $k$-mers | # potential edge $k$-mers | Init. time (fold change) |
|---|---|---|---|---|
| ERR233214_1 | 41,766,273 | 1,134,617 | 6,310,923 | 1.632x |
| SRR1031159_1 | 29,937,099 | 632,996 | 1,088,645 | 1.162x |
| SRR514250_1 | 442,498,904 | 6,656,205 | 53,063,633 | 1.813x |
| SRR553460 | 196,863,538 | 12,271,956 | 38,806,654 | 2.453x |
| Chr15 | 70,240,374 | 1 | 2 | 0.909x |

**Table 3. Two-sided *k*BF overhead.** The number of extra edge $k$-mers that are stored for two-sided *k*BF is compared to the total number of $k$-mers. The one-time initialization overhead includes keeping track of all potential edge $k$-mers and extra time to query which are the true edge $k$-mers. The number of potential edge $k$-mers is compared to the number of true edge $k$-mers as well as the total number of $k$-mers. The initialization time for two-sided *k*BF is shown as a fold-change over populating the classic Bloom filter with the $k$-mer set (average of 10 runs).

| Accession | # $k$-mers Classic | # $k$-mers best match | # $k$-mers hitting set |
|---|---|---|---|
| ERR233214_1 | 41,766,273 | 21,783,670 | 23,635,764 |
| SRR1031159_1 | 29,937,099 | 15,120,795 | 16,992,976 |
| SRR514250_1 | 442,498,904 | 237,264,629 | - |
| SRR553460 | 196,863,538 | 102,224,726 | 115,593,454 |
| Chr15 | 70,240,374 | 36,064,290 | 39,152,979 |

**Table 4. Number of $k$-mers selected by Sparse *k*BF.** Comparison of the number of $k$-mers in the sparsified $k$-mer set for the different implementation methods and for the classic Bloom filter.

ization (measured as the maximum resident set size of the process) is also compared in Table 5.

The relaxed *contains* function, which must be used when $K$ is selected using the hitting set formulation, needs to check more possible neighbouring $k$-mers, making the hitting set sparsification queries slower than the other implementations. The hitting set implementation also does not do as good a job of sparsifying the original set of $k$-mers. Hitting set sparsification also takes orders of magnitude more memory and time than the other methods and than the non-sparse *k*BF implementation.

In contrast to the hitting set sparsification, best match sparsification achieves close to one half of the original $k$-mer set with little extra overhead in initialization time or memory. The strict *contains* function for sparse *k*BF also has a better FPR than the relaxed version and takes less time to perform one million queries. In practice, there is little difference between the best match sparsification and single sequence sparsification, since they will both yield approximately the same $k$-mer set in a case where single sequences are being sparsified. These results mean that best match sparsification is the simplest and best way to sparsify any set of sequences, without having to determine whether it is a special case of non-overlapping sequences.

| | Initialization memory overhead (GB) | | | Initialization time overhead (sec) | | |
|---|---|---|---|---|---|---|
| Accession | 2-$k$BF | Best match | Hitting set | 2-$k$BF | Best match | Hitting set |
| ERR233214_1 | 3.45 | 4.26 | 42.00 | 121.9901 | 192.7540 | 857.8472 |
| SRR1031159_1 | 1.62 | 2.07 | 28.67 | 21.7156 | 21.5318 | 373.4421 |
| SRR514250_1 | 29.54 | 38.41 | - | 1342.1470 | 1856.5640 | - |
| SRR553460 | 17.85 | 22.55 | 198.18 | 699.4796 | 932.5057 | 6012.9880 |
| Chr15 | 3.94 | 4.49 | 67.04 | 43.5708 | 25.7702 | 844.1708 |

**Table 5. sparse *k*BF overhead** Comparison of the one-time overhead for the initialization of the sparse $k$BF implementations. The one time cost of splitting the sequences into $k$-mers, choosing the $k$-mer set, checking the edge $k$-mers, and inserting them into the Bloom filter is reported. The hitting set implementation for SRR514250_1 used up all available memory and did not complete running. Results are the averages over 10 runs.

## 5. Conclusion

Together, the possibilities of drastically reducing the false positive rate or reducing the size of the Bloom filter have the potential to enable continued performance improvements in many applications that use Bloom filters to store $k$-mers from sequences. These performance improvements are necessary to allow biological sequence applications to continue to scale to larger and many more experiments.

**Acknowledgments.** The authors want to thank Dr. Geet Duggal and Hao Wang for the many helpful discussions. This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4554 to Carl Kingsford, by the US National Science Foundation (CCF-1256087, CCF-1319998) and by the US National Institutes of Health (R21HG006913, R01HG007104). C.K. received support as an Alfred P. Sloan Research Fellow.

## References

1. Benoit, G., Lemaitre, C., Lavenier, D., Drezen, E., Dayris, T., Uricaru, R., Rizk, G.: Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. BMC Bioinformatics 16(1), 288 (2015)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
3. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2004)
4. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology 8(22), 1 (2013)
5. Heo, Y., Wu, X.L., Chen, D., Ma, J., Hwu, W.M.: BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. Bioinformatics pp. 1354–1362 (2014)
6. Holley, G., Wittler, R., Stoye, J.: Bloom Filter Trie - a data structure for pan-genome storage. Proceedings of WABI 2015, vol. 9289, pp. 217–230. Springer (2015)

7. Malde, K., O'Sullivan, B.: Using Bloom filters for large scale gene sequence analysis in Haskell. In: Lecture Notes in Computer Science, vol. 5418, pp. 183–194. Springer (2009)
8. Marçais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics 27(6), 764–770 (2011)
9. Patro, R., Mount, S.M., Kingsford, C.: Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. Nature Biotechnology 32(5), 462–464 (2014)
10. Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J.M., Brown, C.T.: Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. Proceedings of the National Academy of Sciences 109(33), 13272–13277 (2012)
11. Rozov, R., Shamir, R., Halperin, E.: Fast lossless compression via cascading Bloom filters. BMC Bioinformatics 15(Suppl 9), S7 (2014)
12. Salikhov, K., Sacomoto, G., Kucherov, G.: Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In: Lecture Notes in Computer Science, vol. 8126, pp. 364–376. Springer (2013)
13. Shi, H., Schmidt, B., Liu, W., Müller-Wittig, W.: Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009). pp. 1–8. IEEE (2009)
14. Solomon, B., Kingsford, C.: Large-scale search of transcriptomic read sets with Sequence Bloom Trees. bioRxiv p. 017087 (2015)
15. Song, L., Florea, L., Langmead, B.: Lighter: fast and memory-efficient sequencing error correction without counting. Genome Biology 15(11), 1–13 (2014)
16. Stranneheim, H., Käller, M., Allander, T., Andersson, B., Arvestad, L., Lundeberg, J.: Classification of DNA sequences using Bloom filters. Bioinformatics 26(13), 1595–1600 (2010)
17. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biology 15(3), R46 (2014)
18. Yu, Y.W., Yorukoglu, D., Berger, B.: Traversing the k-mer landscape of ngs read datasets for quality score sparsification. In: Research in Computational Molecular Biology. pp. 385–399. Springer (2014)
19. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Research 18(5), 821–829 (2008)