Efficient counting of k-mers in DNA sequences using a bloom filter

By Páll Melsted and Jonathan K Pritchard, 2011

Itay Shamir, Yiftach Ginger

January 2019
1 Problem
   - Problem

2 Bloom Filter
   - The data structure
   - False Positive Rate

3 The Algorithm
   - Higher Coverage Cutoffs
   - Parallelism

4 Results
   - Implementation
   - 1000 Genomes Project Pilot II dataset

5 Compares to other algorithms
   - Benchmarks
   - Cuckoo Filters

6 Summary

7 Questions
We’ve seen lots of methods that manipulate k-mers and de-bruijn graphs and they all required large amounts of space.

This is because that even using various compression and representation schemes, most of these methods hold the entire set of the k-mers in one sort of hash-table or another. This requires a lot of space and if we want to increase our coverage of the genome.
Lots of k-mer, lots of space

- Sequencing methods tend to introduce errors into the k-mers
- Errors are random
- A lot of unique faulty k-mers
Lots of k-mer, lots of space

By removing these unique k-mers at the start of the processing we can reduce the space requirements dramatically.

- Team working on the Giant Panda genome was able to reduce their space requirements by removing 68% of the k-mers, from 8.62B to 2.69B just by removing the unique k-mers.
Errors tend to occur at the ends of the reads, but if we ignore this and the possibility of recurring duplicate errors we can model the number of errors as:

\[ GC \frac{l - k + 1}{l} (1 - (1 - \alpha)^k) \]

- \( G \): Length of Genome
- \( C \): Sequence coverage
- \( \alpha \): Uniform error rate
- \( k \): Length of k-mers
- \( l \): Length of reads
Error estimation

For error rate of 1% per base (which is in the error rate range of real instruments), and read length of 100 bp, and $k = 30$

$$\frac{100 - 30 + 1}{30} \left(1 - (1 - 0.01)^{30}\right) = 0.6$$

60% of k-mers are spurious!
Our Goals

Finding the unique (or n-unique) k-mers is time and space consuming. We want to propose a method to find n-unique k-mers that is:

1. Fast
2. Requires little space
3. Tolerates false negatives but not false positives
New data structure

Bloom Filter

- Array of size $m$
- Set of $d$ hash functions
- Inserting and querying: $\forall h : A[h(x)] = 1$
False positives
False positives

\[a\]

\[
\begin{array}{ccccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
False positives

\[ a \] \rightarrow 0 1 0 1 1 0 0 0 0 0 0 \rightarrow b \]
False positives

- Diagram showing that both a and b can have the same bit pattern (0101101010).

The diagram illustrates how Bloom Filters can lead to false positives, where elements that are not actually in the set can be incorrectly identified as being in the set.
False positives

The diagram illustrates a Bloom filter with three elements, $a$, $b$, and $c$. Each element maps to a series of bits in the filter. For example, $a$ maps to the bits [0, 1, 0, 1], $b$ to [0, 1, 1, 0], and $c$ to [0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0]. This shows how Bloom filters can have false positives, where some elements incorrectly indicate membership in the set.
False Positive Rate Approximation

\[
\frac{1}{m} \Rightarrow 1 - \frac{1}{m} \Rightarrow \left(1 - \frac{1}{m}\right)^d \Rightarrow \\
\left(1 - \frac{1}{m}\right)^{nd} \Rightarrow 1 - \left(1 - \frac{1}{m}\right)^{nd} \Rightarrow \\
\left[1 - \left(1 - \frac{1}{m}\right)^{nd}\right]^d \approx \left(1 - e^{-\frac{dn}{m}}\right)^d
\]
false_positive_rate = \left( 1 - e^{-\frac{dn}{m}} \right)^d

We see: tradeoff between the false positive rate and the size of our memory m. We want: given desirable memory size to optimize number of hash functions.

\begin{align*}
g(d) &= d \ln \left( 1 - e^{-\frac{dn}{m}} \right) \\
\frac{\partial g}{\partial d} &= \ln \left( 1 - e^{-\frac{dn}{m}} \right) + \frac{dn}{m} \frac{e^{-\frac{dn}{m}}}{1 - e^{-\frac{dn}{m}}} \\
d &= \frac{m}{n} \ln 2 \Rightarrow \text{false_positive_rate} = 2^{-d}
\end{align*}
The Bloom Filter Principle

The Bloom filter principle - Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.
The Algorithm

Efficient counting of k-mers in DNA sequences using a bloom filter

Itay Shamir, Yiftach Ginger

January 2019
The Algorithm

Efficient counting of k-mers in DNA sequences using a bloom filter

January 2019
The Algorithm

Efficient counting of k-mers in DNA sequences using a bloom filter

January 2019
The Algorithm

Itay Shamir, Yiftach Ginger

Efficient counting of k-mers in DNA sequences using a bloom filter

January 2019
The Algorithm
The Algorithm

- The Bloom Filter algorithm is efficient for counting k-mers in DNA sequences.
- It uses a bloom filter to reduce memory usage and speed up the process.
- The algorithm involves checking if a k-mer is in the bloom filter and, if not, adding it to the hash table.
The Algorithm

Itay Shamir, Yiftach Ginger

Efficient counting of k-mers in DNA sequences using a Bloom Filter

January 2019

24 / 45
The Algorithm

Efficient counting of k-mers in DNA sequences using a bloom filter

January 2019 25 / 45
The Algorithm

![Diagram showing the Bloom Filter algorithm for counting k-mers in DNA sequences.](image)
Canonical k-mer

All k-mers are generated sequentially from the sequencing reads. We do not need to distinguish between a k-mer and its reverse complement sequence.

**canonical k-mer** - whichever of the two versions is lexicographically smaller.

canonical\_k\_mer := \text{min}(x, \text{reversed}(x))
canonical\_k\_mer(ACG) = canonical\_k\_mer(GCA) = ACG
### The Algorithm

**Algorithm 1** Bloom filter k-mer counting algorithm

1. $B \leftarrow$ empty Bloom filter of size $m$
2. $T \leftarrow$ hash table
3. **for all** reads $s$ **do**
4.     **for all** k-mers $x$ in $s$ **do**
5.         $x_{rep} \leftarrow \min(x, \text{revcomp}(x))$ // $x_{rep}$ is the canonical k-mer for $x$
6.         **if** $x_{rep} \in B$ **then**
7.             **if** $x_{rep} \notin T$ **then**
8.                 $T[x_{rep}] \leftarrow 0$
9.             **else**
10.                 add $x_{rep}$ to $B$
11. **for all** reads $s$ **do**
12.     **for all** k-mers $x$ in $s$ **do**
13.         $x_{rep} \leftarrow \min(x, \text{revcomp}(x))$
14.         **if** $x_{rep} \in T$ **then**
15.             $T[x_{rep}] \leftarrow T[x_{rep}] + 1$
16. **for all** $x \in T$ **do**
17.     **if** $T[x] = 1$ **then**
18.         remove $x$ from $T$

---

- **Inserting non-unique k-mers to $T$**
- **Counting non-unique k-mers**
- **Remove false positives**
Some applications may be required a higher coverage cutoff. Either to filter out sequencing errors or to simply extract sequences of interest.

**Extension**

- Assume coverage $c$
- Each bit in the bit array replaced with a counter with size $\lceil \log_2 c \rceil$
- When inserting to the bloom filter: $B[h_i(x)] \leftarrow \min\{B[h_i(x)], c - 1\}$
- To check if a k-mer should be inserted into the hash table $T$ we look to see if all of $B[h_i(x)]$ are equal to $c - 1$. 
Parallelism

Possible to speed up the operations using multiple cores with lock-free data structures.

Require:

- Non-blocking implementation of the hash table.
- Extending the Bloom filter to return the number of bits set to 1 when querying, and the number of bits changed from 0 to 1 when inserting.
Example

Process 1

query: x

return: 2

insert: x

return: 1

Update hash table

Bloom Filter
3-hash functions.

Process 2

query: x

return: 2

insert: x

return: 0

Not update hash table
https://web.stanford.edu/group/pritchardlab/bfcounter.html - git not very active but updated last year (created 8 years ago).

- store a 1-byte counter for each k-mer and by default k-mers take 8-bytes of memory with a maximum k of 31 (although if desired, larger k-mers can be specified at compile time)

- Require the user to specify an estimate for the number of k-mers, and use a Bloom filter with 4 times as many bits as the expected number of k-mers.

- This corresponds to a memory usage of 4-bits per k-mer and the optimal number of hash functions functions for the Bloom filter is $d = 3$. 
Illumina dataset

- 7.5M 100bp paired-end read.
- Corresponds to approximately 32-fold coverage of Chromosome 21
- $k = 31$

Out of 80.4M observed k-mers, slightly more than half (48.7M) are observed only a single time. The vast majority of these singleton k-mers (99.87%) are not found in the reference genome.
Illumina dataset
2.66 billion 36bp paired-end reads.

40-fold coverage of individual NA19240.

$k = 25.$

<table>
<thead>
<tr>
<th>Program</th>
<th>Time (hrs)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFCounter</td>
<td>23.82</td>
<td>42</td>
</tr>
<tr>
<td>Jellyfish</td>
<td>8.03</td>
<td>71</td>
</tr>
<tr>
<td>Naive</td>
<td>&gt;26.38</td>
<td>&gt;128</td>
</tr>
</tbody>
</table>

Part of the difference in speed is due to BFCounter taking a second pass through the data to obtain exact k-mer counts (which may not be essential for all applications).
1000 Genomes Project Pilot II dataset

- There are 12.18 billion k-mers in the sequencing reads. 9.35 billion are unique (77%) and 2.83 billion have coverage of two or greater.
- About 0.5 billion of the unique k-mers were stored in the hash table after the first phase which corresponds to a 5.3% false positive rate.
- BFCounter stored 27% of the original k-mers after the first pass, and this was cut to 23% after false positives were removed.
Time vs Reads

Memory vs k-mers

![Graph comparing memory usage of different algorithms](image)

- khmer (1% false positive rate)
- khmer (5% false positive rate)
- khmer (20% false positive rate)
- Tallymer
- Jellyfish
- DSK
- KMC
- BFCounter
- scTurtle
- KAnalyze
Disk vs k-mers

![Graph showing benchmarks for different algorithms.

- khmer (1% false positive rate)*
- khmer (1% false positive rate), gzip-compressed
- Tallymer
- Jellyfish
- DSK
- KMC
- BFCounter
- scTurtle
- KAnalyze

**Disk usage (GB)**

**Total number of distinct k-mers (billions)**

*Itay Shamir, Yiftach Ginger
Efficient counting of k-mers in DNA sequences using a Bloom Filter
January 2019 39 / 45

- Among the in-memory algorithms, BFCounter utilized the smallest amount of memory.

Cuckoo Filters

- Array of $m$ Buckets.
- Each bucket hold $b$ values.
- Two hash functions.

Given input:

- Try to insert to the bucket that determine by the first hash (succeed if not full).
- If fail, try the second bucket that determine by the second hash, if bucket is full - then the occupant of bucket 2 is evicted and the value is placed there.
Cuckoo Filters

(a) before inserting item $x$
(b) after item $x$ inserted
(c) A cuckoo filter, two hash per item and functions and four entries per bucket
Cuckoo Filters vs Bloom Filters

- Offering deletion.
- Slower insertion.
- Faster querying.
- Space complexity - depend on the desire false positive rate, better for low (< 0.5%).

Usually in DNA sequencing we have relative high false positive rate (in both filters), and bloom filter is more space-efficient in those settings.
Summary

- Removing unique k-mers can lead to huge reduction in numbers of k-mers.
- Bloom filters are great for reducing memory if we can tolerate false positives.
- Treadoff between memory, false positive rate and run time.
- Empirical results.
- Comparisons to other algorithms.
Questions

- What is the problem that we try to solve using bloom filter? and how it is used to solve it?
- Assume that given sequencing has 6% error rate and 65% of the k-mers were removed by the counting process (assume all errors removed), and assume for simplicity that there was 1 k-mer per read, what was the most probable k (as in k-mer) that was used?
- Assume you have 1385 k-mers that you want to count, and have 10000 bits of memory for the bloom filter, how many hash functions you will use to minimize the false positive rate, and what this rate will be?
- Assume you have the following bloom filter: \( m = 5, k = 3 \), the hash functions are: \( f_1(x) = x \mod 5 \), \( f_2(x) = x \mod 3 \), where \( x \) is the representation of the k-mer as a 3-digit number, with the key: \( A = 1, C = 2, G = 3, T = 4 \), i.e. the k-mer ACG = 123 and then \( f_1(ACG) = 123 \mod 5 = 3 \). Given the above definition, show a BFCounter run for the following K-mers: AAA, ACG, GCC, TGT, GCA, GAG

Send answers to: itayshamir@tau.mail.ac.il, yi.ginger@gmail.com