Bioinformatics, 35(23), 2019, 4871–4878
doi: 10.1093/bioinformatics/btz299
Advance Access Publication Date: 30 April 2019
Original Paper

OXFORD

Genome analysis

kmcEx: memory-frugal and retrieval-efficient encoding of counted *k*-mers

Peng Jiang[1,†], Jie Luo[1,†], Yiqi Wang[1], Pingji Deng[1], Bertil Schmidt[2], Xiangjun Tang[1], Ningjiang Chen[3], Limsoon Wong[4,*] and Liang Zhao[1,3,*]

[1]Precision Medicine Research Center, Taihe Hospital, Hubei University of Medicine, Shiyan, Hubei 442000, China, [2]Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz 55128 Germany, [3]School of Computing and Electronic Information, Guangxi University, Nanning, Guangxi 530004, China and [4]School of Computing, National University of Singapore, Singapore 117417, Singapore

*To whom correspondence should be addressed.
†The authors wish it to be known that, in their opinion, the first two authors should be regarded as Joint First Authors.
Associate Editor: Bonnie Berger

Received on June 29, 2018; revised on April 2, 2019; editorial decision on April 16, 2019; accepted on April 19, 2019

Abstract

Motivation: *K*-mers along with their frequency have served as an elementary building block for error correction, repeat detection, multiple sequence alignment, genome assembly, etc., attracting intensive studies in *k*-mer counting. However, the output of *k*-mer counters itself is large; very often, it is too large to fit into main memory, leading to highly narrowed usability.
Results: We introduce a novel idea of encoding *k*-mers as well as their frequency, achieving good memory saving and retrieval efficiency. Specifically, we propose a Bloom filter-like data structure to encode counted *k*-mers by coupled-bit arrays—one for *k*-mer representation and the other for frequency encoding. Experiments on five real datasets show that the average memory-saving ratio on all 31-mers is as high as 13.81 as compared with raw input, with 7 hash functions. At the same time, the retrieval time complexity is well controlled (effectively constant), and the false-positive rate is decreased by two orders of magnitude.
Availability and implementation: The source codes of our algorithm are available at github.com/lzhLab/kmcEx.
Contact: wongls@comp.nus.edu.sg or s080011@e.ntu.edu.sg
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Next-generation sequencing (NGS) technologies have evolved so fast that the cost of generating a single human's whole genome has been pushed down to 100 USD by Illumina NovaSeq systems (www.illumina.com). This high-throughput and low-cost technology is useful for unveiling some mysteries of life via data manipulations (Schaid *et al.*, 2018). Among them, *k*-mer has served as an elementary yet essential building block.

A *k*-mer is a short substring of a sequencing read having *k* consecutive bases. Besides the *k*-mer itself, its frequency is also very important for data analysis, which is defined as the counts that it appears in input data. For instance, the frequency of the 3-mer 'ACA' in the read 'AACATACACG' is 2. We use frequency and count interchangeably.

K-mers have been widely used in genome assembly (Zerbino and Birney, 2008), error correction (Zhao *et al.*, 2017), repeat detection (Marçais and Kingsford, 2011), sequence clustering (James *et al.*, 2018), etc. For example, they have been adopted to build the de Bruijn graph (Compeau *et al.*, 2011) for genome assembly, where the nodes of the graph are the *k*-mers and the weights of the edges are the frequencies, e.g. velvet (Zerbino and Birney, 2008).

Counting the frequency of *k*-mers from a set of input reads is simple in principle: (i) sliding each read from the beginning to the end via a *k*-length window and; (ii) accumulating the count of each

© The Author(s) 2019. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

k-mer as it is encountered. However, considering the huge memory required to store all the k-mers, several disk-/memory-based approaches have been developed, including BFCounter (Melsted and Pritchard, 2011), DSK (Rizk *et al.*, 2013), khmer (Crusoe *et al.*, 2015), KMC3 (Kokot *et al.*, 2017) and KCMBT (Mamun *et al.*, 2016). Although k-mer counting can be achieved by these approaches, a more challenging yet less studied problem emerges: how the k-mers as well as their frequency can be encoded frugally and efficiently so that they can be used freely at will.

Loading k-mers as well as their frequency into main memory directly is not a good idea. The key reason is that the huge data size is often beyond the capacity of a typical computer's main memory, not to mention using the data. For example, the output of all 31-mers having frequency no less than 2 is 90G for the HapMap sample NA12878 (https://www.ncbi.nlm.nih.gov/sra/ERR091571/). A barely acceptable solution is to load partial data into main memory and automatically swap it with disk-based data when needed. Clearly, this strategy slows down running time. Although various algorithms have been developed to compress biological sequences, they are unable to tell the existence of a query (a k-mer) directly, i.e. decompression is usually needed (Hosseini *et al.*, 2016). Another memory-economical k-mer representation approach is Bloom filter based, which has been heavily used in de Bruijn graph-based sequence assembly (Bowe *et al.*, 2012; Chikhi *et al.*, 2014; Chikhi and Rizk, 2013; Pandey *et al.*, 2017; Pell *et al.*, 2012; Salikhov *et al.*, 2014). Nonetheless, these approaches only consider k-mers themselves but not their frequency, except deBGR (Pandey *et al.*, 2017), which uses counting quotient filter to cope with frequency. To tackle this problem, we have conceived a novel idea to encode k-mers as well as their frequency and have achieved resource economy and retrieval efficiency.

We introduce a Bloom filter-like data structure (Bloom, 1970), named as kmcEx, having two bit arrays instead of one and take the order of the hash functions (used in Bloom filter) into account. Among the two bit arrays, one is used to record k-mers while the other is for representing frequency of k-mers. Hence, both the k-mers as well as their frequency can be recorded frugally. This is the core of our innovative approach. Bloom filter is resource efficient, and together with a simple k-mer separation trick, we can achieve 13.8 times smaller memory usage by default settings and retrieve a k-mer and its frequency in effectively constant time while reducing false-positive rate (FPR) by two orders of magnitude. Our proposed algorithm can be integrated into downstream data analysis procedures to narrow down the 'next-generation gap' (McPherson, 2009), e.g. sequence assembly and variants calling.

2 Algorithm: encoding and decoding

2.1 k-mer encoding

Let $\mathbb{K}$ be a set of k-mers (obtained from sequencing reads), and $\mathbb{F}$ be the set of frequencies of the corresponding k-mers in $\mathbb{K}$, i.e.

$$\mathbb{F} = \{f_\kappa | \kappa \in \mathbb{K}\},$$

where f_κ is the total count of κ in the set of given reads. Suppose the maximum frequency in $\mathbb{F}$ is $f_{\max}$, i.e. $f_{\max} = \max(\mathbb{F})$, and it can be represented by at most h bits (in binary). The encoding of $\mathbb{K}$ and $\mathbb{F}$ can be achieved by the following steps:

1. Create h hash functions, say $\{H_0(\cdot), H_1(\cdot), \cdots, H_{h-1}(\cdot)\}$, and keep them in a vector H.
2. Allocate a coupled-bit array $B = (B^+, B^-)$ having m bits. m is computed as $m = -n \ln p / (\ln 2)^2$ with the optimal number of hash functions h as well as the target FPR p, and n is the number of k-mers to be encoded; *cf.* Bloom (1970).
3. For a k-mer κ in $\mathbb{K}$, set the corresponding bits of B indexed by H as

$$\begin{aligned} B^+[H_i(\kappa)] &= 1, & i &\in (0, 1, \ldots, h-1), \\ B^-[H_i(\kappa)] &= \text{Binary}(f_\kappa)^h[i], & i &\in (0, 1, \ldots, h-1), \end{aligned}$$

where $\text{Binary}(f_\kappa)^h$ is the binary representation of f_κ by h bits, and $\text{Binary}(f_\kappa)^h[i]$ returns the value of i-th bit. For instance, $\text{Binary}(50)^7 = 0110010$, and $\text{Binary}(50)^7[2] = 1$.

4. Repeat Step 3 above until all k-mers and their frequency are encoded.

Following the above procedure, both $\mathbb{K}$ and $\mathbb{F}$ can be stored in coupled-bit arrays; see Algorithm 1. For ease of understanding, we have just outlined the main idea but overlook some details here, including collision control and frequency binning; however, they will be presented with full details in the next section shortly.

2.2 k-mer decoding

Decoding a k-mer, say κ, from the coupled-bit array B is straightforward:

1. Check whether all the h bits of B^+ hashed by H are set, i.e. $\forall i \in \{0, 1, \ldots, h-1\}\ B^+[H_i(\kappa)] == 1$. If all the h bits are set to '1', then proceed to Step 2 to retrieve the frequency f_κ, otherwise return 'False', meaning that κ is not contained in B.

Algorithm 1: Encoding k-mers as well as their frequency.

```
Data: K: k-mers; F: frequencies of K; H: hash function vector
Result: L: a list of (B+, B−)
begin
    f_max ← max(F)
    h ← |Binary(f_max)|
    while K ≠ ∅ do
        initialize new B+, B− and K′
        for κ in K do
            flag ← False
            freq ← Binary(f_κ)
            Roll-back point                                   *
            for i ← 0 to h − 1 do
                j ← H_i(κ)
                if B+[j] == 1 and B−[j] ≠ freq[i] then
                    flag ← True
                    K′ ← K′ ∪ {κ}
                else
                    B+[j] ← 1
                    B−[j] ← freq[i]
                if flag == True then
                    roll back B+ and B− to the point *
                    break;
        L ← L ∪ {(B+, B−)}
        K ← K′
    return L
```

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

Algorithm 2: Decoding a k-mer and its frequency.

Data: κ: a k-mer; B: the coupled-bit array; H: hash function vector
Result: f_κ: the frequency of κ
begin
  $h \leftarrow |H|$
  for $i \leftarrow 0$ to $h-1$ **do**
    if $B^+[H_i(\kappa)] == 0$ **then**
      return False
  for $i \leftarrow 0$ to $h-1$ **do**
    $b_i \leftarrow B^-[H_i(\kappa)]$
  val $\leftarrow$ Denary$(b_0 b_1 \cdots b_{(h-1)})$
  return val

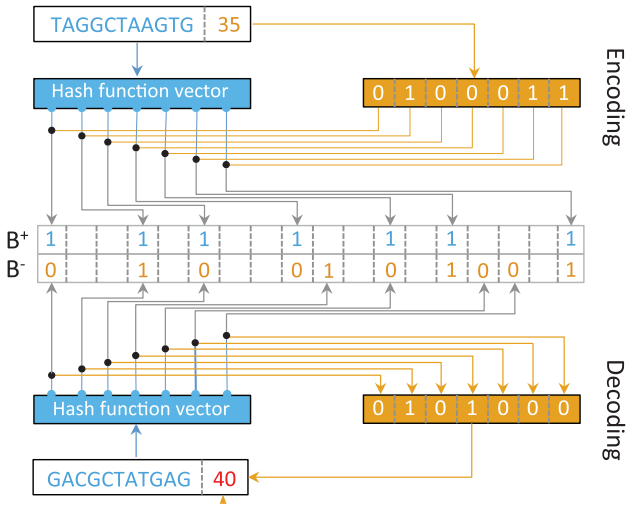

Fig. 1. An example of encoding and decoding a k-mer as well as its frequency by using a coupled-bit array

2. Determine the frequency of κ by

$$f_\kappa = \text{Denary}(B^-[H_0(\kappa)]B^-[H_1(\kappa)]\cdots B^-[H_{h-1}(\kappa)]),$$

where Denary$(\cdot)$ transforms the binary represented number into the decimal mode. For instance, Denary$(0100011) = 35$.

The detailed procedure is shown in the Algorithm 2. Similarly, for ease of understanding we have ignored how false positives are controlled here but put it in the next section. Obviously, decoding the frequency of a k-mer from a coupled-bit array is in constant time complexity. The overall encoding and decoding diagram is illustrated by an example shown in Figure 1.

3 Algorithm: other details

We have described the main framework of the algorithm, i.e. encoding and decoding. Now it remains to discuss some other details, viz. collision control, false-positive reduction, frequency binning and k-mer separation.

3.1 Collision control

The insertion of items into a traditional Bloom filter does not take care of collisions because the state of each bit only changes from '0' to '1'. Hence, all the items can be inserted. However, the situation changes in our approach, in which an additional bit array B^- is used. Unlike B^+, the states of B^- could be changed without control. That is, once a bit changes from '0' to '1', it could be changed back to '0' again. When uncontrolled, this leads to wrong frequency recording. Technically, a collision happens when the following condition is met: there is $i \in \{0, 1, \ldots, h-1\}$ such that

$$B^+[H_i(\kappa)] == 1 \text{ and}$$
$$B^-[H_i(\kappa)] \neq B'^-[H_i(\kappa)],$$

where κ is the k-mer to be inserted and B' is the updated coupled-bit array after κ is inserted.

To control collisions during k-mer insertion, we allow multiple coupled-bit arrays to be created; i.e. once a collision happens when a k-mer is being inserted to the current coupled-bit array, we defer the insertion to a new coupled-bit array to be created later; see Algorithm 1. In this way, collisions can be avoided.

Theoretically, the optimal bit array size (say m) is given by $m = -n \ln p/(\ln 2)^2$, where n is the number of k-mers inserted and p is the FPR. Since n is decreased after some numbers of k-mers have been inserted to existing coupled-bit arrays, m becomes smaller. Based on our experiments, three to four coupled-bit arrays are sufficient to hold all the k-mers as well as their frequency, even for very large data, such as whole-genome sequencing reads of human; see Results for more details.

3.2 False-positive reduction

False positives generally exist when decoding a k-mer and its frequency from a coupled-bit array B due to the overlaid use of bits. Owing to false positives, two unwanted situations may occur: (i) a nonexistent k-mer is called mistakenly from a coupled-bit array and; (ii) an existent k-mer is found in multiple coupled-bit arrays. Although the FPR is usually small, typically less than 0.1%, it is still harmful to downstream data analysis. To reduce FPR, we introduce an idea of checking the existence of a k-mer's neighbors, say $N(\kappa)$, as well as its $(k$-2$)$-mer (the canonical form of the middle one, viz. $\kappa_{2:(k-1)}$). We define $N(\kappa)$ as

$$N(\kappa) = \{x \cdot \kappa_{[1:(k-1)]}\} \cup \{\kappa_{[0:(k-2)]} \cdot x\}, x \in (A, C, G, T),$$

where κ is a k-mer, $\kappa_{[i:j]}$ returns the substring of κ from i to j with both ends included, and the $\cdot$ operation concatenates the two strings at both sides. Regarding $(k$-2$)$-mers, they are stored in a classical Bloom Filter, say B_{k-2}.

In case no k-mer in $N(\kappa)$ can be found from existing coupled-bit arrays or its $(k$-2$)$-mer is absent from B_{k-2}, then κ is considered as a false positive to be discarded; hence 'scenario (i)' is solved. Analogously, if κ is found in more than one coupled-bit arrays, we choose the closest $f_{\kappa'}$ as the frequency of f_κ, where κ' is a k-mer in $N(\kappa)$, i.e. a neighbor of κ. In this way, 'scenario (ii)' can be solved. The rationale is that the frequency of a k-mer's neighbors should be very close to the frequency of the k-mer itself. Statistically, the probability of a false positive's neighbors also contain some false positives is about 1% ($=\sum_{\kappa' \in N(\kappa)} p$, typically $p = 0.1\%$ and $|N(\kappa)| = 8$). Hence, the FPR is reduced by two orders of magnitude.

Our idea of using neighbor checking is similar to that introduced in Pellow *et al.* (2017). However, we also consider frequency to reduce FPRs. In this study, we use one-side neighborhood checking,

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

hence no false negative is introduced unless the length of a sequencing read is exactly k.

Note that, two Bloom filters are used to hold (k-2)-mers having frequency of 1 and larger than 1, respectively. This is because the two types of k-mers are encoded by a classical Bloom filter and coupled-bit arrays separately. Since this two types of data structure have various FPRs, dealing with them separately leads to fine-grained control of FPR reduction.

3.3 Frequency binning

One may notice that the number of hash functions h is unnecessarily the same as the minimum number of bits to represent the maximum frequency, i.e. f_{max}. We keep this intentionally for achieving optimal trade-off between memory-saving ratio and FPR. According to Swamidass and Baldi (2007), the estimated number of items contained in a Bloom filter bit array is

$$n^* = -\frac{m}{h}\ln\left(1-\frac{X}{m}\right),$$

where m is the bit array size, h is the number of hash functions and X is the number of bits set to '1'. Clearly, increasing h decreases n^*. Hence, we need to decrease h to insert more k-mers into a bit array, thus the higher ratio of memory saving. On the other hand, the Bloom filter (Bloom, 1970) theory says the relation between p (the FPR), m (the bit-array size), n (the number of items inserted in the bit array) and h (the number of hash functions) is:

$$p = \left(1-\left(1-\frac{1}{m}\right)^{hn}\right)^h \approx \left(1-e^{-\frac{hn}{m}}\right)^h.$$

Obviously, increasing h decreases the FPR. Hence, by considering both aspects we can fix an optimal h that generates high memory-saving ratio as well as low FPR via the above equations. Unfortunately, the fixed h may not be large enough to represent the largest frequency f_{max}.

To solve this problem, we propose a novel idea of clustering the frequencies partially. Specifically, we keep the frequencies that are lower than a threshold unchanged, and the rest are grouped into several bins. Mathematically, the frequencies are grouped into 2^h bins: each of the first 2^{h-2} bins keeps exactly the same frequency as the index of that bin (1-based); the next 2^{h-1} bins use equal width binning (EWB) at width of 3; and the last 2^{h-2} bins use EWB covering all the rest frequencies. This can be expressed by the following formula:

$$\mathrm{Bin}[i] = \begin{cases} i, & \text{if } \ i \le 2^{h-2} \\ [(i-1):(i+1)], & \text{if } \ 2^{h-2} < i \le 2^{h-1}+2^{h-2} \\ \mathrm{EWB}^{2^{h-2}}(\{i | i > b_0, i \in \mathbb{F}\})[i-b_0], & \text{others} \end{cases}$$

where $\mathbb{F}$ is the set of all the frequencies, $\mathrm{EWB}^{2^{h-2}}(\{X\})$ clusters X into 2^{h-2} groups by EWB algorithm, i.e. the bin width is

$$\text{bin width} = \frac{f_{max} - (2^{h-2}+2^{h-1})}{2^{h-2}},$$

$\mathrm{EWB}(\cdot)[i]$ returns the items in the ith bin, and $b_0 = 2^{h-2}+2^{h-1}$. For instance, suppose $h=7$ and the frequencies range from 1 to 1000, then the first 32 bins record the frequencies exactly as they are, the next 64 bins keep frequency from 33 to 224 at bin width of 3, and the last 32 bins hold all the rest frequencies at bin width of 24.

The clustered frequency of each bin is defined as the index of the bin plus the half bin width in integer.

The rationale of different bin widths to achieve better performance is based on the following widely accepted facts. (i) The k-mers having very low frequency are either containing sequencing errors or from the hardly sequenced regions; hence keeping the frequency exactly is crucial for these k-mers. (ii) The k-mers having very high frequency are very likely from repeat regions, where the exact frequency itself is not so important; hence a close value reflecting its frequent level is informative enough. (iii) The k-mers having frequency close to the normal coverage are less informative, hence their frequency can be altered by 1 without affecting the performance of subsequent data analysis. For instance, we can use 60 to represent all the k-mers having frequency of 59, 60 and 61 without affecting downstream data analysis. We demonstrate that the impact of binning is negligible when h is large enough (typically 7, even for human NGS reads; details are shown in Section 4).

3.4 k-mer separation

Instead of using coupled-bit arrays to encode k-mers having frequency of 1, we use a classical Bloom filter to represent them. That is, by fixing a FPR (p, usually set to 0.1%) we allocate a bit array having $m = -n\ln p/((\ln 2)^2$ bits to store all these k-mers, assuming there are n single-occurrence k-mers. This is similar to that of conventional Bloom filter-based k-mer representation approaches (Chikhi *et al.*, 2014; Salikhov *et al.*, 2014). However, frequencies are considered in our model, while they are ignored in these conventional approaches.

Regarding k-mers having frequency larger than 1, they are encoded via the proposed approach.

4 Results

4.1 Datasets

Five real NGS datasets are collected to validate the effectiveness of the proposed algorithm for encoding k-mers as well as their frequency: *Staphylococcus aureus* (D1), *Rhodobacter sphaeroides* (D2), human chromosome 14 (D3), Bombus impatiens (D4) and the HapMap sample NA12878 (D5). The first four datasets are from GAGE (Salzberg *et al.*, 2012) which contains the gold standard data for NGS analysis evaluation. The last one is from the NCBI sequence read archive. The five datasets represent different complexities and scales that are commonly encountered in NGS data analysis. Details are shown in Table 1 and the Supplementary Notes.

4.2 Good memory-saving ratio achieved

We use KMC3 (Kokot *et al.*, 2017) to generate all the k-mers as well as their frequency, and apply our algorithm to encode the output. Experiments show that the proposed algorithm effectively encodes the k-mers along with their frequency, achieving a memory-saving ratio of 13.81 ± 2.86 on average for the five datasets comparing to the corresponding raw input size, when $h=7$, $k=31$ and frequency≥1; see Figure 2 and Table 2. Here the raw input size of a set of counted k-mers is the size they occupied on a disk without applying any compression technique. With the same parameters, the average FPR is as low as (9.04 ± 4.37)e-4, and the average absolute distance (AAD) between the real frequency and the clustered frequency is 2.23e-3 ± 9.75e-4. The FPR is computed as N_{dif}/N_{all}, where N_{dif} is the number of k-mers whose decoded frequency is different from its encoded frequency and N_{all} is the total number of k-mers. The AAD is computed as $\frac{1}{N}\sum_{\kappa\in\mathbb{F}}||f_\kappa - f'_\kappa||$, where N is the total number of k-mers and f'_κ is the clustered frequency of κ.

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

Table 1. The datasets used for *k*-mers encoding capability examination

Dataset	Genome size	Read length	Coverage	No. paired-end reads	Input size (fastq)
D1	2.8M	101	46.3×	1 294 104	280M
D2	4.6M	101	33.6×	766 646	446M
D3	88.3M	101	38.3×	16 757 120	9.4G
D4	249.2M	124	150.8×	303 118 594	92G
D5	3121.8M	101	27.6×	854 084 773	442G

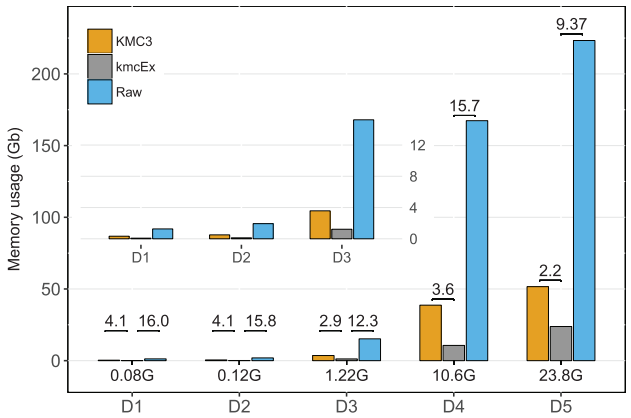

Fig. 2. Memory usage comparison between kmcEx, KMC3 and the raw input. The number over each pair of bars shows the ratio of memory usage between the two approaches, while the number under a gray bar is the real memory usage of kmcEx. For the sake of clarity, the results of D1, D2 and D3 are enlarged in the inset figure. Results shown here are obtained at $k = 31$, $h = 7$ and frequency≥1

Taking D5 (the largest dataset) as an example, the input raw sequencing reads is 442G, and all 31-mers (including those having frequency of 1) occupy 223.3G space. Using our approach, it only takes 23.83G to store all the *k*-mers as well as their frequencies, achieving a memory-saving ratio of 9.37. In addition, the FPR is 1.52e-3 and the AAD is 2.63e-3.

For *k*-mers having frequency larger than 1, we achieve average memory-saving ratio of 5.55 ± 0.15 and FPR of 2.84e-4 ± 1.83e-5. The small standard deviation of memory-saving ratio and FPR indicate that our algorithm is stable regardless of input size and complexity.

Since KMC3 has the ability to represent *k*-mers as well as their frequency in compressed form, we also compare the memory-saving ratio between our approach and KMC3. Based on all *k*-mers (including those having frequency of 1), KMC3 takes 3.40 ± 0.84 times more memory than our approach on the five datasets at $k = 31$ and $h = 7$. See Figure 2 for more details.

4.3 Smaller *h* increases memory-saving ratio

The number of hash functions, *h*, is a key factor affecting memory-saving ratio. To examine its impact, we have conducted experiments for *h* equals 6, 7 and 8. Results show that a smaller *h* results in higher memory-saving ratio but larger FPR. In the contrast, larger *h* yields lower memory-saving ratio but smaller FPR. Specifically, the average memory-saving ratio of 31-mers of the five datasets is 14.61 ± 2.67, 13.81 ± 2.86, 13.05 ± 2.94 for *h* equals 6, 7 and 8, respectively. The average FPR is (2.74 ± 1.37)e-3, (9.04 ± 4.37)e-4, (1.35 ± 0.87)e-4 for *h* equals 6, 7 and 8, respectively. More details are shown in Figure 3.

Even when $h = 6$, the FPR is still very small. For example, the FPR is 4.17e-3 for D5 when $h = 6$ and $k = 31$. However, small *h* leads to increased AAD. Again, taking D5, the AAD is 1.27e-2 for $h = 6$, but it is strikingly reduced to 6.95e-3 and 5.0e-4 for $h = 7$ and 8, respectively. This result suggests that one can achieve high memory-saving ratio via decreasing *h* for data analysis when frequency is not critical. But generally, $h = 7$ is a very good trade-off.

4.4 Larger *k* elevates memory-saving ratio

Usually *k* varies with the complexity of the input; i.e. a larger *k* is required to cope with more complicated genomes, while a smaller *k* is sufficient for simpler genomes. For instance, a typical *k* used for analyzing human genome is 31, while 23 is chosen given *S.aureus* genome as input.

To examine the impact of *k* on the memory-saving ratio, we carried out experiments on all the datasets having *k* ranging from 23 to 31. Results obtained from k-mers having frequency greater than 1 show that larger *k* yields higher memory-saving ratio. The average ratio when $h = 7$ is 4.14 ± 0.09, 4.49 ± 0.04, 4.84 ± 0.07, 5.20 ± 0.07, 5.55 ± 0.15 for *k* equals 23, 25, 27, 29 and 31, respectively. The detailed results are shown in the Figure 4. We exclude *k*-mers having frequency of 1 when examining the impact of *k*. This is because these *k*-mers are encoded by a classical Bloom filter, which produces very similar memory-saving ratio when parameters are the same (see Table 3).

Intuitively, increasing *k* gives rise to exponentially more possible *k*-mers and correspondingly needs exponentially more space to hold all these possible *k*-mers. However, in practice, most of the possible *k*-mers do not actually occur in the reads. For example, 2 568 720 383 23-mers occur in D5 while 2 750 560 119 31-mers occur in D5, an increase of 7% (even though there would be 4^{31-23} times more possible 31-mers than possible 23-mers). The allocated bit array size is proportional to the number (but not length) of *k*-mers that actually occur in the reads. Hence memory used is not increased much when *k* increases. Correspondingly, memory-saving ratio is not increased much when *k* increases; cf. Figure 4. In fact, memory-saving ratio is observed to be correlated linearly with *k* (correlation coefficient $r = 0.999$). For instance, the ratio of memory-saving ratios between $k = 31$ and $k = 23$ is $5.55/4.14 = 1.34$, while the ratio of the length of 31-mers and 23-mers is $31/23 = 1.35$. This is not surprising. Recall that the size m_k of a coupled-bit array is proportional to the number n_k of *k*-mers to be encoded, viz. $m_k = \alpha n_k$ where $\alpha = -\ln p/(\ln 2)^2$. Thus, the ratio of the two memory-saving ratios is $m_{31}/m_{23} = (\alpha n_{31}/\alpha n_{23}) = n_{31}/n_{23} \approx 31/23$ as observed.

4.5 Reduction of false-positive rate

Both neighborhood checking and (*k*-2)-mer Bloom filter are introduced to reduce false positives.

By using neighborhood checking, the FPR is markedly reduced by at least two orders of magnitude on coupled-bit arrays. Generally, a larger number of hash functions produces a lower FPR, and using longer *k*-mers yields better FPR reduction; see Figure 5. Precisely, the average ratio of reduced FPR to that of the original one is 0.0247 ± 0.0088, 0.0199 ± 0.0083, 0.0204 ± 0.0059 for *h*

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

Table 2. Memory-saving ratios of *k*-mer encoding for all *k*-mers on the five datasets

Data	*k*	23			25			27			29			31		
	h	6	7	8	6	7	8	6	7	8	6	7	8	6	7	8
D1		16.32	15.51	15.01	16.33	15.52	15.02	16.30	15.75	14.99	16.21	15.67	15.15	16.38	15.82	15.04
D2		15.97	15.27	14.48	16.21	15.51	14.86	16.32	15.62	14.97	16.38	15.67	15.01	16.38	15.67	15.02
D3		13.02	12.04	11.19	13.15	12.18	11.31	13.26	12.27	11.44	13.34	12.37	11.54	13.41	12.45	11.63
D4		16.01	15.24	14.53	16.16	15.41	14.73	16.28	15.54	14.86	16.36	15.64	14.97	16.43	15.72	15.06
D5		10.00	8.942	8.081	10.16	9.097	8.236	10.28	9.213	8.352	10.36	9.299	8.438	10.431	9.368	8.500

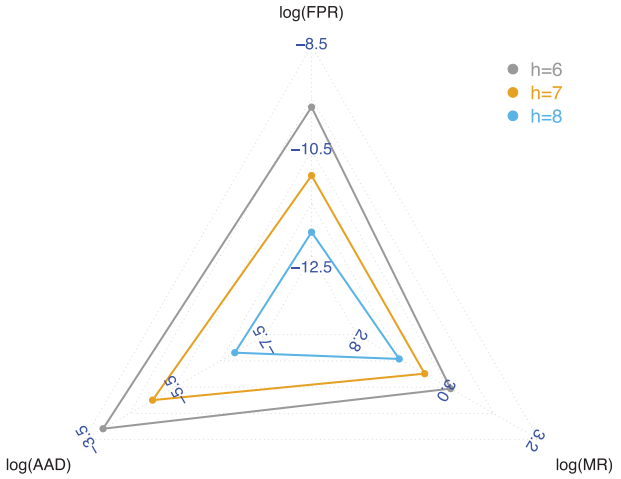

Fig. 3. The effect of number of hash functions *h* to false-positive rate (FPR), averaged absolute distance (AAD) and memory-saving ratio (MR). The data shown here is the mean value obtained from the five real datasets for each metrics having $k = 31$ and frequency≥1. For ease of reading, the axes are shown in log scale having base of *e*

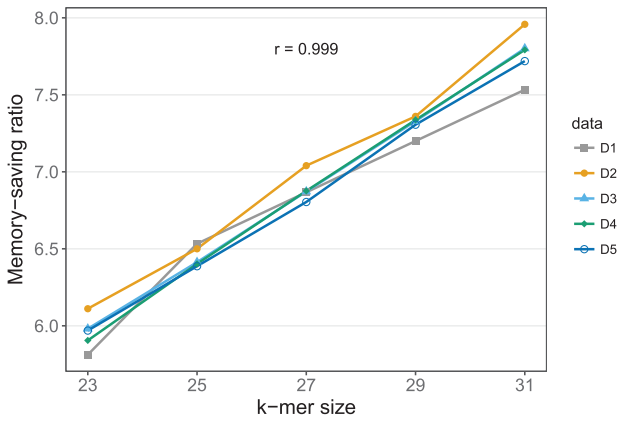

Fig. 4. The correlation between memory-saving ratio and the size *k* when $h = 7$ and frequency≥2

equals 6, 7 and 8, respectively. In terms of *k*, this value is 0.0247 ± 0.0076, 0.0223 ± 0.0083, 0.0201 ± 0.0078, 0.0193 ± 0.0080, 0.0187 ± 0.0071 for *k* sets to 23, 25, 27, 29 and 31, respectively. In addition, we have also examined the performance on different datasets, and found that all the datasets generate similar results except D4. By further checking D4 (details are shown in the Table 1), we found that its coverage is significantly higher than others. We speculate that the deep coverage introduces more erroneous *k*-mers that have noteworthy frequency, thus harder to distinguish. Our guess has been validated via reducing the coverage of D4.

Applying (*k*-2)-mer Bloom filter on all *k*-mers including those having frequency larger than 1 as well as neighborhood checking, we are able to reduce the FPR by about two orders of magnitude on average. Note that, to achieve good memory efficiency, we have decreased the number of hash functions in constructing both Bloom filters.

4.6 Negligible difference of frequency via binning

Frequency binning is used to achieve higher memory-saving ratio while controlling FPR as well as the discrepancy of frequency. Experiments show that this has negligible impact on the frequencies after binning. On average, the AAD is 0.0198 ± 0.0031, 0.0067 ± 0.0047, 0.0011 ± 0.0017 for *h* is 6, 7 and 8, respectively. Clearly, the number of hash functions has great impact on the discrepancy of frequency. Typically, seven hash functions are good enough to represent all the frequencies. We also calculated the correlation between *k*-mer size and AAD, and found that the correlation coefficient is −0.122 and the *P*-value is 0.845, indicating that they have no significant relationship.

Taking D3 as an example, its *k*-mer frequency distribution is shown in Figure 6. With seven hash functions, the difference between the real and modified frequency is 0 for those smaller than 32; the largest difference is 1 for those greater than 32 but less than 224 and; the maximum difference is 12 for those higher than 224. Obviously, these discrepancies have marginal impact on downstream data analysis. Hence, it is quite acceptable to adopt this strategy.

4.7 Separating *k*-mers improves memory-saving ratio

K-mers having frequency of 1 form a large proportion. For instance, the number of 31-mers in D5 having frequency of 1 is 4.17 billion, while the number of 31-mers having frequency larger than 1 is 2.75 billion. Instead of treating these *k*-mers by coupled-bit arrays, we use a classical Bloom filter to represent these *k*-mers. Results show that the memory-saving ratio can be significantly elevated. Taking $k = 31$, $h = 7$ and $p = 0.1\%$ (predefined FPR for a Bloom filter) as an example, the average memory-saving ratio for classical bit array is 20.3 ± 0.18 and, this value for coupled-bit array is 5.6 ± 0.15. The overall memory-saving ratio is 13.8 ± 2.86. Detailed results are shown in Table 3.

Note that, although a large portion of *k*-mers having frequency of 1 contains errors, around 14% of these are error-free (data obtained from the five datasets except D4). In addition, taking erroneous *k*-mers into account is also important for sequencing error correction.

4.8 Running speed analysis

All experiments are conducted on a super computer having 256G RAM and two E5-2683V4 CPUs (32 cores in total), installed with CentOS 7.0.

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

Table 3. Memory-saving ratio dissection for 31-mers when $h=7$

Data	$\|k\text{-mer}_1\|$(m)	$\|k\text{-mer}_{2-1000}\|$(m)	MU_1(G)	MU_{2-1000}(G)	MR_1	MR_{2-1000}	MR_{all}
D1	35.67	3.49	0.056	0.021	20.53	5.599	15.82
D2	54.13	5.91	0.085	0.035	20.52	5.689	15.67
D3	372.09	99.92	0.593	0.626	20.22	5.378	12.45
D4	4643.11	543.89	7.418	3.227	20.17	5.678	15.72
D5	4171.45	2748.5	6.665	17.16	20.17	5.396	9.368

Note: MU, memory usage; MR, memory-saving ratio. The subscript '1' means *k*-mers having frequency of 1, while the subscript '2–1000' represents *k*-mers having frequency lager than 1 but less or equal to 1000. Note, the number of *k*-mers having frequency larger than 1000 is negligible, and their frequency is usually replaced by 1000 by default for *k*-mer counters.

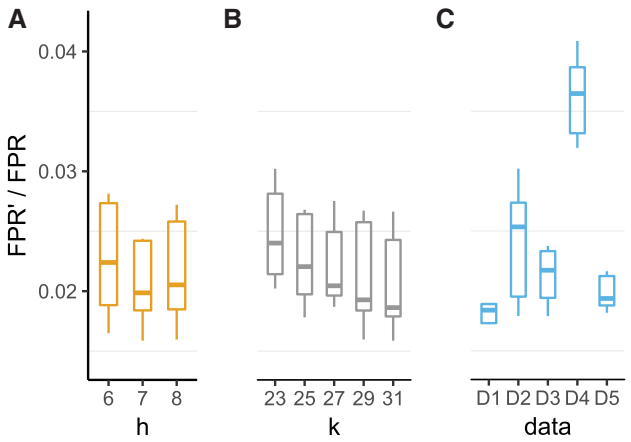

Fig. 5. A comprehensive comparison of FPR reduction via joint examination on *k*-mers having frequency≥2. Panel (**A**) shows the FPR reduction w.r.t. the number of hash functions (*h*). Panel (**B**) is the relation between the reduction and the *k*-mer size (*k*) and Panel (**C**) reveals the reduction on different datasets. The 'FPR' is the original false-positive rate, while the 'FPR'' is the reduced FPR obtained when the neighbors of a *k*-mer and the (*k*-2)-mer are jointly considered

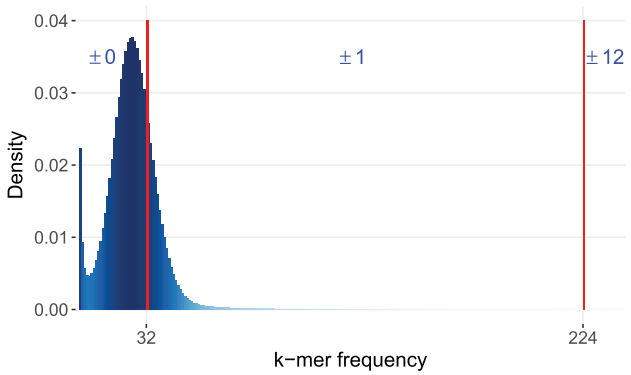

Fig. 6. The frequency distribution of *k*-mers having value larger than 2. The two vertical red lines indicate the boundaries of different binning approaches, and the blue numbers are the largest absolute difference between the real frequency and the modified frequency. The data are produced from D3 having *k* of 25

Both the encoding and decoding procedure are implemented in multithread. For encoding, multiple threads are applied to construction of coupled-bit arrays but not Bloom filters; while for decoding, all queries can be multithreaded. Results show that the average encoding speed is 1.31 m/s (million *k*-mers per second), while the average decoding speed is 0.51 m/s and 0.69 m/s for present and absent *k*-mers, respectively. Detailed results are shown in Table 4.

To speed up queries, we use a hash table to store *k*-mers that are unable to be inserted into five coupled-bit arrays although more coupled-bit arrays can be allocated. Typically, the hash table is small.

The FPR as well as false-negative rate of kmcEx are also explored. Results show that the false-negative rate is 0 for present *k*-mers. Regarding FPR, the average value is 9.04e-4 for absent *k*-mers. See details in Table 4.

Querying both present and absent *k*-mers from kmcEx is feasible as shown in experiments above. However, one may argue that enumerating *k*-mers from a Bloom filter-like data structure, e.g. kmcEx, is unfeasible since the 'string' of a *k*-mer is not stored explicitly. In reality, this is not a big issue as has been discussed elsewhere (Pandey *et al.*, 2017; Salikhov *et al.*, 2014). Firstly and most importantly, *k*-mers stored in kmcEx is obtained from reads, meaning that *k*-mers are known from input; second, the *k*-mers have been generated as input for the construction of the Bloom filter; they can be compressed, stored on disk and reproduced from disk when needed (this works because they only need to be enumerated).

Comparing running speed of our approach and khmer (Crusoe *et al.*, 2015), khmer is much slower than our approach in *k*-mer encoding, and even in fact cannot yield results in reasonable time duration. For instance, the time of encoding 31-mers by khmer is 2.55 times longer than our approach on D4, and it keeps running for several hours on D5. In addition, the FPR as well as frequency distortion of khmer are markedly greater than our approach. The advantage of khmer is that it is faster than our approach in *k*-mer querying. For instance, on average it takes 0.748 s for our approach to query 500k presented *k*-mers; whereas, the time costed by khmer is 0.173 s. More detailed results are shown in the Supplementary Notes.

Comparing query speed with KMC3 (Kokot *et al.*, 2017), our approach is several times slower. Regarding database opening time, our approach is markedly faster than KMC3. See details in Table 4 and the Supplementary Notes. Inspired by the elegant data structure of KMC3, we will dedicate to query speed improvement in future work.

5 Concluding remarks

Manipulating *k*-mers and their frequency has been an essential yet elementary task for many real applications. Many approaches have been developed to count *k*-mers effectively. However, the huge amount of *k*-mers generated from a typical NGS dataset has posted a great challenge for data analysis. Hence, we present a novel *k*-mer encoding and decoding algorithm based on a Bloom filter-like data structure. We have achieved a noteworthy memory-saving ratio of about 14 folds between raw memory usage and compressed memory usage. Meanwhile, we are able to reduce FPR an order of magnitude

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020

Table 4. Running time of encoding and decoding on the five datasets having $k=31$, $h=7$ and frequency≥ 1

Data	Encoding		Decoding							
	|k-mers| (m)	Time (s)	|k-mers| (k)	Present				Absent			
				$Time_{open}$ (s)	$Time_{query}$ (s)	FPR	FNR	$Time_{open}$ (s)	$Time_{query}$ (s)	FPR	FNR
D1	39.2	14.6	500	0.269	1.005	0	0	0.241	0.671	4.8e-4	0
D2	60.0	24.2	500	0.304	0.846	0	0	0.224	0.589	5.2e-4	0
D3	472.0	238.2	500	3.049	0.974	0	0	3.107	0.738	8.6e-4	0
D4	5187.0	2152.9	500	19.28	1.197	0	0	20.78	0.701	1.1e-3	0
D5	6919.1	3969.3	500	86.35	1.222	0	0	88.15	1.062	1.5e-3	0

Note: FPR, false-positive rate; FNR, false-negative rate; encoding and decoding are run by four threads. Note that the opening time of query is the whole time of loading all the encoded k-mers of a dataset.

and keep fidelity of k-mer frequency. The decoding process is also well controlled so that its time complexity is effectively constant.

Funding

This study was collectively supported by the Natural Science Foundation of China [31501070, 81671831, 61762008 and 81702482]; the Natural Science Foundation of Hubei [2017CFB137] and Guangxi [2016GXNSFCA380006 and 2018GXNSFAA281275], China; the Scientific Research Found of Guangxi University [XGZ150316] and Taihe Hospital [2016JZ11]. Additionally, L.W. was supported, in part, by a Kwan Im Thong Hood Cho Temple chair professorship.

Conflict of Interest: none declared.

References

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

Bowe,A. *et al.* (2012) Succinct de Bruijn graphs. In: Raphael,B. and Tang,J. (eds) *Algorithms in Bioinformatics*. Springer, Berlin, Heidelberg, pp. 225–235.

Chikhi,R. and Rizk,G. (2013) Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithm. Mol. Biol.*, **8**, 22.

Chikhi,R. *et al.* (2014) On the representation of de Bruijn graphs. In: Sharan, R. (ed.) *Research in Computational Molecular Biology*. Springer International Publishing, Cham, pp. 35–55.

Compeau, P.E.C. *et al.* (2011) How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.*, **29**, 987–991.

Crusoe,M.R. *et al.* (2015) The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, **4**, 900.

Hosseini,M. *et al.* (2016) A survey on data compression methods for biological sequences. *Information*, **7**, 56.

James,B.T. *et al.* (2018) MeShClust: an intelligent tool for clustering DNA sequences. *Nucleic Acids Res.*, **46**, e83.

Kokot,M. *et al.* (2017) KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, **33**, 2759–2761.

Mamun,A.-A. *et al.* (2016) KCMBT: a k-mer counter based on multiple burst trees. *Bioinformatics*, **32**, 2783–2790.

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

McPherson,J.D. (2009) Next-generation gap. *Nat. Methods*, **6**, S2–S5.

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

Pandey,P. *et al.* (2017) deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, **33**, i133–i141.

Pell,J. *et al.* (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl. Acad. Sci. USA*, **109**, 13272–13277.

Pellow,D. *et al.* (2017) Improving Bloom filter performance on sequence data using k-mer Bloom filters. *J. Comput. Biol.*, **24**, 547–557.

Rizk,G. *et al.* (2013) DSK: k-mer counting with very low memory usage. *Bioinformatics*, **29**, 652.

Salikhov,K. *et al.* (2014) Using cascading Bloom filters to improve the memory usage for de Bruijn graphs. *Algorithm. Mol. Biol.*, **9**, 2.

Salzberg,S.L. *et al.* (2012) GAGE: a critical evaluation of genome assemblies and assembly algorithms. *Genome Res.*, **22**, 557–567.

Schaid,D.J. *et al.* (2018) From genome-wide associations to candidate causal variants by statistical fine-mapping. *Nat. Rev. Genet.*, **19**, 491–504.

Swamidass,S.J. and Baldi,P. (2007) Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *J. Chem. Inform. Model.*, **47**, 952–964.

Zerbino,D.R. and Birney,E. (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, **18**, 821–829.

Zhao,L. *et al.* (2017) MapReduce for accurate error correction of next-generation sequencing data. *Bioinformatics*, **33**, 3844–3851.

Downloaded from https://academic.oup.com/bioinformatics/article-abstract/35/23/4871/5481953 by Eccles Health Sci Lib-Serials user on 03 July 2020