Lighter: fast and memory-efficient error correction without counting

Li Song[1] Liliana Florea[2] Ben Langmead*[3]

April 26, 2014

Abstract

Lighter is a fast and memory-efficient tool for correcting sequencing errors in high-throughput sequencing datasets. Lighter avoids counting k-mers in the sequencing reads. Instead, it uses a pair of Bloom filters, one populated with a sample of the input k-mers and the other populated with k-mers likely to be correct based on a statistical test. As long as the sampling fraction is adjusted in inverse proportion to the dataset's average coverage, the Bloom filter size can be held constant while maintaining near-constant accuracy. Lighter is easily applied to very large sequencing datasets. It is parallelized, uses no secondary storage, and is both faster and more memory-efficient than competing approaches while achieving comparable accuracy. Lighter is free open source software available from https://github.com/mourisl/Lighter/.

Introduction

The cost and throughput of DNA sequencing have improved rapidly in the past several years [6], with recent advances reducing the cost of sequencing a single human genome at 30-fold coverage to around $1,000 [7]. With these advances has come an explosion of new software for analyzing large sequencing datasets. Sequencing error correction is a basic need for many of these tools. Removing errors at the outset of an analysis can improve accuracy of downstream tools such as variant callers [12]. Removing errors can also improve the speed and memory-efficiency of downstream tools, particularly for analyses involving de novo assembly with De Bruijn graphs since the graph's size increases with the number of distinct k-mers in the dataset [3, 19].

To be useful in practice, error correction software must make economical use of time and memory even when input datasets are extremely large (billions of reads) and when the genome under study is also large (billions of nucleotides). Several methods have been proposed, covering a wide tradeoff space between accuracy, speed and memory- and storage-efficiency. SHREC [21] and HiTEC [9] build a suffix index of the input reads and locate errors by finding instances where a substring is followed by a character less often than expected. Coral [20] and ECHO [11] find overlaps among reads and use the resulting multiple alignments to detect and correct errors. Reptile [24] and Hammer [15] detect and correct errors by examining each k-mer's neighborhood in the dataset's k-mer Hamming graph.

The most practical and widely used error correction methods descend from the spectral alignment approach introduced in the earliest De Bruijn graph based assemblers [3,19]. These methods count the number of times each k-mer occurs (its *multiplicity*) in the input reads, then apply a threshold such that reads with multiplicity exceeding the threshold are considered *solid*. Solid k-mers are unlikely to have been altered by sequencing errors. k-mers with low multiplicity (*weak* k-mers) are systematically edited into solid k-mers using a dynamic-programming solution to the spectral alignment problem [3,19] or, more often, a fast heuristic approximation. Quake [12], the most widely used error correction tool, uses a hash-based k-mer counter called Jellyfish [14] to determine which k-mers are solid. CUDA-EC [22] was the first to use a Bloom filter as a space-efficient alternative to hash tables for counting k-mers and for representing the set of solid k-mers. More recent tools such as Musket [13] and BLESS [8] use a combination of Bloom filters and hash tables to count k-mers or to represent the set of solid k-mers.

Lighter (LIGHTweight ERror corrector) is also in the family of spectral alignment methods, but differs from previous approaches in that it avoids counting k-mers. Rather than count k-mers, Lighter samples k-mers randomly, storing the sample in a Bloom filter. Lighter then uses a statistical test applied to each position of each read to compile a set of solid k-mers, stored in a second Bloom filter. These two Bloom filters are the only sizable data structures used by Lighter.

Lighter's crucial advantage is that its parameters can be set such that the memory footprint and the accuracy of the approach are near-constant with respect to coverage. That is, no matter how deep the coverage, Lighter can allocate the same sized Bloom filters and achieve nearly the same (a) Bloom filter occupancy, (b) Bloom filter false positive rate, and (c) error correction accuracy. Lighter does this without using any disk space or other secondary memory. This is in contrast to BLESS and Quake/Jellyfish, which use secondary memory to store some or all of the k-mer counts. Lighter's accuracy is comparable to competing tools, as we show both in simulation experiments where false positives and false negatives can be measured, and in real-world experiments where read alignment scores and assembly statistics can be measured. Lighter is also very simple and fast, faster than all other tools tried in our experiments. These advantages make Lighter quite practical compared to previous counting-based approaches, all of which require an amount of memory or secondary storage that increases with depth of coverage.

Method

Lighter's workflow is illustrated in Figure 1. Lighter makes three passes over the input reads. The first pass obtains a sample of the k-mers present in the input reads, storing the sample in Bloom filter A. The second pass uses Bloom filter A to identify solid k-mers, which it stores in Bloom filter B. The third pass uses Bloom filter B and a greedy procedure to correct errors in the input reads.

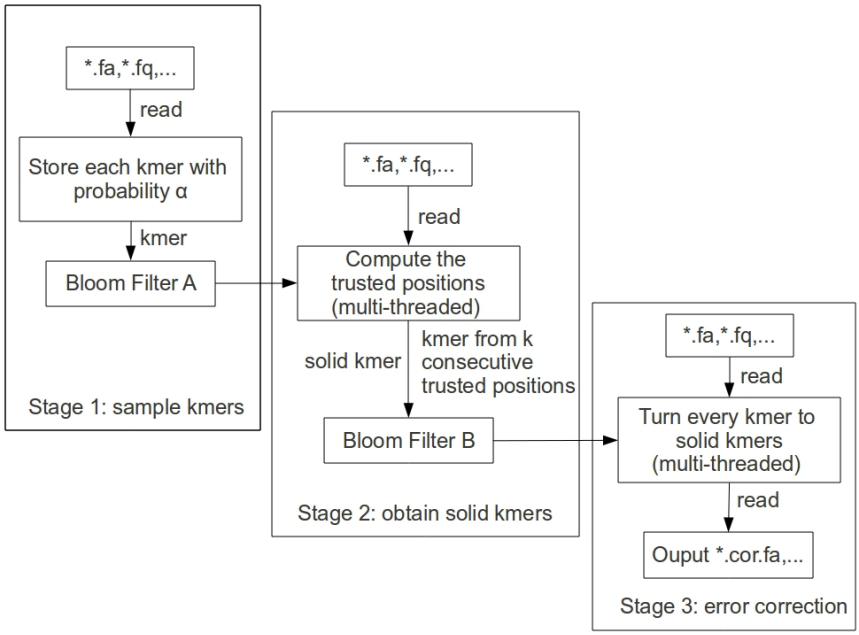

Figure 1: The framework of Lighter

Bloom filter

A Bloom filter [1] is a compact probabilistic data structure representing a set. It consists of an array of m bits, initialized to 0. To add an object o, h independent hash functions $H_0(o), H_1(o), ..., H_{h-1}(o)$ are applied. Each maps o to an integer in $[0, m)$ and the corresponding h array bits are set to 1. To test if object q is a member, the same hash functions are applied to q. q is a member if all corresponding bits are set to 1. A false positive occurs when the corresponding bits are set to 1 "by coincidence," that is, because of objects besides q that were added previously. Assuming the hash functions map objects to bit array elements with equal probability, the Bloom filter's false positive rate is approximately $(1 - e^{-h\frac{n}{m}})^h$, where n is the number of distinct objects added, which we call the *cardinality*. Given n, which is usually determined by the dataset, m and h can be adjusted to achieve a desired false positive rate. Lower false positive rates can come at a cost, since greater values of m require more memory and greater values of k require more hash function calculations. Many variations on Bloom filters have been proposed that additionally permit compression of the filter, storage of count data, representation of maps in addition to sets, etc [23]. Bloom filters and variants thereon have been applied in various bioinformatics settings, including assembly [18], compression [10], k-mer counting [17], and error correction [22].

By way of contrast, another way to represent a set is with a hash table. Hash tables do not yield false positives, but Bloom filters are far smaller. Whereas a Bloom filter is an array of bits, a hash table is an array of buckets, each large enough to store a pointer, key, or both. If chaining is used, lists associated with buckets incur additional overhead. While the Bloom filter's small size comes at the expense of false positives, these can be tolerated in many settings including in error correction.

As Figure 1 shows, the efficiency of bloom filter affects the running time of Lighter a lot. For a standard Bloom filter, the h hash functions may map o to any places in the bit array. The bit array is usually very large, so all the h accesses will likely to cause cache miss. In [cite], the authors proposed blocked Bloom filter which can decrease the number of cache misses. Given a block size b, it will use $H_0(o)$ to decide the start position on the bit array, and then map $H_0(o), H_1(o), ..., H_{h-1}(o)$ onto the block starting from that position. It choose b about the size of a cache line, then the h accesses will not cause h cache misses. The drawback of using blocked Bloom filter is that we have to use a bit larger h and m to have the same FPR when using the standard bloom filter. To estimate the FPR of blocked Bloom filter, we can consider each of the possible $m - b + 1$ blocks. for the i-th block, the FPR within this block is $(b'_i/b)^h$, where b'_i is the number of bits set to 1 in block i. So the overall FPR is $\frac{\sum_i (b'_i/b)^h}{m - b + 1}$.

In our method, the objects to be stored in the Bloom filters are k-mers. Because we would like to treat genome strands equivalently for counting purposes, we will always *canonicalize* a k-mer before adding it to, or using it to query a Bloom filter. A canonicalized k-mer is either the k-mer itself or its reverse complement, whichever is lexicographically prior.

Sequencing model

We use a simple model for the sequencing process (including errors) and for Lighter's subsampling. The sequencing model resembles one suggested previously [16]. Let K be the total number of k-mers obtained by the sequencer. We say a k-mer is *incorrect* if its sequence has been altered by one or more sequencing errors. Otherwise it is *correct*. Let ϵ be the fraction of k-mers that are incorrect. We assume ϵ does not vary with the depth of sequencing. The sequencer obtains correct k-mers by sampling independently and uniformly from k-mers in the genome. Let the number of k-mers in the genome be G, and assume all are distinct. If κ_c is a random variable for the multiplicity of a correct k-mer in the input, κ_c is binomial with success probability $1/G$ and number of trials $(1 - \epsilon)K$: $\kappa_c \sim Binom((1 - \epsilon)K, 1/G)$. Since the number of trials is large and the success probability is small, the binomial is well approximated by a Poisson: $\kappa_c \sim Pois((1 - \epsilon)K/G)$

A sequenced k-mer survives subsampling with probability α. If κ'_c is a random variable for the number of times a correct k-mer appears in the subsample, $\kappa'_c \sim Binom((1 - \epsilon)K, \alpha/G)$, which is approximately $Pois(\alpha(1 - \epsilon)K/G)$.

The model for incorrect k-mers is similar. The sequencer obtains incorrect k-mers by sampling independently and uniformly from k-mers "close to" a k-mer in the genome. We might define these as the set of all k-mers with low Hamming distance from some genomic k-mer. If κ_e is a random variable for the multiplicity of an incorrect k-mer, κ_e is binomial with success probability $1/H$ and number of trials ϵK: $\kappa_e \sim Binom(\epsilon K, 1/H)$, which is approximately $Pois(\epsilon K/H)$. It is safe to assume $H \gg G$, but we need not specify H precisely here. $\kappa'_e \sim Pois(\alpha\epsilon K/H)$ is a random variable for the number of times an incorrect k-mer appears in the subsample.

Others have noted that, given a dataset with deep and uniform coverage, incorrect k-mers occur rarely while correct k-mers occur many times, proportionally to coverage [3, 19].

Stages of the method

First pass. In the first pass, Lighter examines each k-mer of each read. With probability $1-\alpha$, the k-mer is ignored. Otherwise, it is canonicalized and added to Bloom filter A. The subsampling fraction α is set by the user.

Say a distinct k-mer K occurs a total of N_K times in the dataset. If subsampling discards all N_K occurrences, the k-mer is never added to A and A's cardinality is reduced by one. Thus, reducing α can in turn reduce A's cardinality. Because correct k-mers are more numerous, incorrect k-mers tend to be discarded from A before correct k-mers as α decreases. Later we show that if K increases but αK is held constant, the cardinality of A and the fraction of correct k-mers in A both remain nearly constant.

Second pass. A read position is overlapped by up to x k-mers, $1 \leq x \leq k$, where x depends on how close the position is to either end of the read. For a position altered by sequencing error, the overlapping k-mers are all incorrect and are unlikely to appear in A. We apply a threshold such that if the number of k-mers overlapping the position and appearing in Bloom filter A is less than the threshold, we say the position is *untrusted*. Otherwise we say it is *trusted*. Each instance where the threshold is applied is called a *test case*. When one or more of the x k-mers involved in two test cases differ, we say the test cases are distinct.

Let $P^*(\alpha)$ be the probability an incorrect k-mer appears in A, including false positives. If random variable $B_{0,x}$ represents the number of k-mers appearing in A for an untrusted position overlapped by x k-mers, $B_{0,x} \sim Binom(x, P^*(\alpha))$. We define thresholds y_x, for each possible value of x in $[1, k]$. Each y_x is the minimum integer such that $p(B_{0,x} \leq y_x - 1) \geq 0.995$.

Initially we ignore false positives and model the probability of a sequenced a k-mer having been added to A as $P(\alpha) = 1 - (1-\alpha)^{f(\alpha)}$. We define $f(\alpha) = max\{2, 0.1/\alpha\}$. That is, we assume the multiplicity of a weak k-mer is at most $f(\alpha)$, which will often be a conservative assumption, especially for small α. It is also possible to define $P(\alpha)$ in terms of random variables κ_e and κ'_e, but we avoid this for simplicity.

A property of this threshold is that when α is small, $P(\alpha/z) = 1-(1-\alpha/z)^{0.1z/\alpha} \approx 1-(1-\alpha)^{0.1/\alpha} = P(\alpha)$, where z is a constant greater than 1 and we use the fact that $(1-\alpha/z)^z \approx 1-\alpha$.

For $P^*(\alpha)$, we additionally take A's false positive rate into account. If the false positive rate is β, then $P^*(\alpha) = P(\alpha) + \beta - \beta P(\alpha)$.

Once all positions in a read have been marked *trusted* or *untrusted* using the threshold, we find all instances where k trusted positions appear consecutively. The k-mer made up by those positions is added to Bloom filter B.

Third pass. In the third pass, a simple, greedy error correction procedure is applied to each read. For a read r of length $|r|$, we have $|r| - k + 1$ k-mers and k_i denotes the k-mer starting at read position i where $1 \leq i \leq |r| - k + 1$. We first identify the longest consecutive stretch of k-mers from the read that appear in Bloom filter B. If the stretch covers the entire read, we conclude there are no errors and make no corrections. Otherwise, let k_b and k_e be the k-mers at the extreme left and right edges of the stretch. If k_e is not the rightmost k-mer in the read, we examine successive k-mers to the right, $k_e + 1$ through $k_{|r|+k+1}$. For each k-mer k_i that does not appear in B, we assume the nucleotide at offset $i+k-1$ is incorrect. We consider all possible ways of substituting for the incorrect nucleotide; for each, we count how many consecutive k-mers

beginning with k_i and stretching to the right now appear in Bloom filter B. We choose the nucleotide that creates the longest stretch of consecutive k-mers in B. If more than one candidate substitution is equally good, we declare position $i+k-1$ to be ambiguous and we resume the scanning process starting with k-mer $k_i + k$. The procedure is illustrated in Figure 2.

If k_b is not the leftmost k-mer in the read, we apply a similar procedure moving to the left.

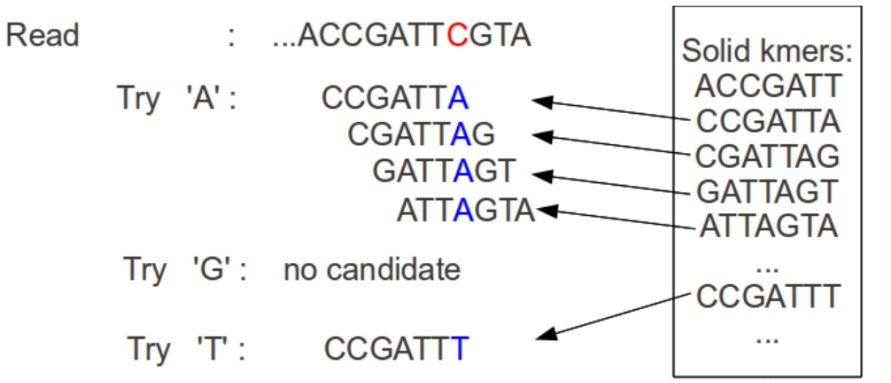

Figure 2: An example of the greedy error correction procedure. k-mer CCGATTC does not appear in Bloom filter B, so we attempt to substitute a different nucleotide for the C shown in red. We select A since it yields the longest stretch of consecutive k-mers that appear in Bloom filter B.

If the error is located near the end of the read, for each candidate substitution, we will extend reads using the kmer reported in Bloom filter A by one path that has the least lexicographical order. Thus we can solve some ties that are more likely to happend near the end of a read due to insufficient extension.

Scaling with coverage

We show we can keep Lighter's memory footprint constant and keep accuracy near-constant as the depth of sequencing K increases. When K increases, we adjust α in inverse proportion. That is, we hold αK constant. If the coverage increases by factor z, we can set $\alpha_2 = \alpha_1/z$, and we will show that it almost keeps the performance of the capturing solid kmers and rejecting weak kmers in the next.

We can show that false positive of Bloom filter A stays almost constant. Suppose we have two scenarios, in scenario I, the sequencer outputs K_1 k-mers and in scenario II, the sequencer outputs zK_1 k-mers. If we set α to be α_1 in scenario I, then $\alpha_2 = \alpha_1/z$ in scenario II. As a result, the distribution of κ'_c and κ'_e do not change in scenario II and are $Pois(\alpha_1(1-\epsilon)K_1)$ and $Pois(\alpha_1\epsilon K_1/H)$ repsectively. The number of correct k-mer get stored in Bloom filter A is $Gp(\kappa'_c \geq 1)$ and the number of incorrector k-mer stored in Bloom filter A is $Hp(\kappa'_e \geq 1)$. These two numbers are the same in scenarios I and II, thus the occupancy rate and the false positive rate of Bloom filter A stays almost the same.

Secondly, we can show that if a position and its nearby positions are error-free then its probability of setting as trusted does not change. We already showed that when α is small, $P(\alpha_1) \approx P(\alpha_2)$ and the FPR of Bloom filter A is almost the same, hence $P^*(\alpha_1) \approx P^*(\alpha_2)$. So the threshold y' differentiating whether a position is trusted or not is constant when changing the coverage. Let $p_1 = \frac{p(\kappa'_c \geq 1)}{p(\kappa_c \geq 1)}$ be the probability of a correct kmer get sampled by both sequencer and Lighter conditioned on it is sampled by the

sequencer. The number of k-mers covering this position in Bloom filter A follows a binomial distribution $B_1 \sim Binom(x, p_1+\beta-\beta p_1)$. In general, $p_1 = \frac{1-e^{-\alpha(1-\epsilon)K/G}}{1-e^{-(1-\epsilon)K/G}} \approx 1-e^{-\alpha(1-\epsilon)K/G}$, which is because $(1-\epsilon)K/G$ is a large number and $e^{-(1-\epsilon)K/G}$ is close to 0. As a result, p_1 does not change in scenario I and II and is about $1-e^{-\alpha_1(1-\epsilon)K_1/G}$. So the binomial distribution B_1 does not change. Moreover, the number of different test cases does not change and the overall positions that get classified as trusted is almost the same when changing the coverage.

Thirdly, we show that the number of positions that contain error and are falsely set as trusted position does not cause trouble. For simplicity, we ignore β, the affect of the false responses from the Bloom filter A, for now. Let $p_2 = \frac{p(\kappa'_e \geq 1)}{p(\kappa_e \geq 1)}$. And the number of sampled k-mers covering this position is roughly a binomial distribution $B_2 \sim Binom(x, p_2)$. In general, $p_2 = \frac{1-e^{-\alpha\epsilon K/H}}{1-e^{-\epsilon K/H}}$. Since $\epsilon K/H$ is very close to 0, $e^{-\epsilon K/H} \approx 1-\epsilon K/H$. So $p_2 \approx \frac{\alpha\epsilon K/H}{\epsilon K/H} = \alpha$. In sceanrio I, $p_2 = \alpha_1$; and in scenario II, $p_2 = \alpha_1/z$. The probability of falsely trusting a position is: $p(B_2 \geq y') = \sum_{i=y'}^{x} \binom{x}{i} p_2^i (1-p_2)^{x-i} \leq \sum_{i=y'}^{x} \binom{x}{i} p_2^i$. Thus the upper bound decreases by a factor much larger than z when we decrease α by factor z. After verifying by simple experiments on the binomial distribution, we conclude that when p_2 is small, $p(B_2 \geq y')$ should decrease by factor no less than z. In other words, the probability of falsely trusting a position is decreased by factor z. After increaing the coverage, we introduce new sequence errors and the total number of different test cases will roughly increase by factor z. Therefore, the number of false trusting will stay the same. Another perspective to get is conclusion is that because the total number of weak kmers get stored in Bloom filter A is constant and the number of different test cases increase, for each test case the number of k-mers we retrieved from Bloom filter A will decrease. When α gets too small, β may dominant the probability. However, in that case, the probability of $B_2 \geq y'$ is extremely small, so it does not cause trouble at that time. For example, in this paper, β is less than 0.004 and the threshold y' is 6 when $\alpha \leq 0.05$ and the k-mer size is 17. Considering the binomial distribution $B'_2 \sim Binom(x, 0.004)$, the probability $p(B'_2 \geq 6)$ is about $5 * 10^{-11}$.

Therefore, the number of k-mers coming from the consecutive trusted positions does not increase which means there is no need to increase the size of Bloom filter B when increasing coverage. Also, the analysis showed that the ability of capturing solid kmers remains almost the same when we change α linearly.

In both table A and B, we only store the cannonical form of a kmer, that is the kmer itself or its reverse complement depending on who has smaller lexicographical order.

In Lighter, user need to specify the value of α. A rule of thumb for choosing α is by applying the nearly linear relationship between $1/\alpha$ and the coverage, and in this paper we choose $\alpha = 0.05$ when the coverage is 70×. So given the coverage C, we can set α to $0.05\frac{70}{C}$.

Quality score

Lighter can also uses the information of quality score. It firstly looks at the first up to 1 million reads of the file, and record the quality score for the last base. Then Lighter will choose the lowest quality score such that at least 5% of the last base's quality scrore from the reads it just looked is no more than it. Then later, when Lighter decides whether a position is trusted or not, if its qulaity score is no more than the threshold quality score, then we set it to untrusted no matter what.

Parallization

As shown in Figure 1, Lighter works in three stages: sampling kmers, obtaining solid kmers and error correction. For the sampling kmers stage, because α is very small, most of the time is spent on scanning the reads namely reading the files. As a result, the overhead of parallilize this part will dominate the benefits, so we decide to remain this stage serial. In the second stage, each thread will handle different reads independently. Reading bloom filter in parallel is very efficient since there is no data race. When a thread find a solid kmer, it firstly tests whether this kmer is stored in table B or not. If this kmer is not in table B,

then we write this kmer into the table. In this way, we can avoid many writing locks for table B because a solid kmer shows up many times. The thrid stage can parallized embarrsingly.

Evaluation

Simulated data set

We compared the our result with other released programs, Quake[cite], Musket[cite] and Bless[cite]. We use Mason[cite] to generate a set of data sets with ecoli as the reference genome. The K12 strain of ecoli is fully sequenced and has the decent length for us to generated different scenarios easily. The k-mer size is 17 for all the programs.

In the simulated data set, we considered the coverage of 35x, 75x, 140x, and the average error rate of 1% and 3%. So there are totally 6 data sets. Because Mason can also specifiy the error rate of the first and the last base repsectively, suppose the average error rate is e, we set the error rate of the first base to $e/2$ and the error rate of the last base is $3e$. This setting is more realistic than a uniform error rate model.

We say a base is true positive(TP) if we identify the error and correct it into the right one; a base is false positive(FP) if we correct an original error-free base; a base is false negative(FN) if it is wrong and we either fail to detect its error or we make a wrong correction. As tradition[cite], we use four measurement to evaluate the error correction methods: recall=TP/(TP+NP), precision=TP/(TP+FP), F-score=2×recall×precision/(recall+precision) which is a kind of average of recall and precision and Gain=(TP-FP)/(TP+FN) which measure the benefit of using error correction.

Coverage		35×		70×		140×	
Error rate		1%	3%	1%	3%	1%	3%
Recall	quake	89.59	48.77	89.64	48.82	89.59	48.78
	musket	92.61	92.04	92.60	92.05	92.60	92.03
	bless	98.68	97.29	98.69	97.48	98.65	97.47
	lighter	**99.42**	**98.03**	**99.36**	**98.93**	**99.39**	**98.99**
Precision	quake	**99.99**	**99.99**	**99.99**	**99.99**	**99.99**	**99.99**
	musket	99.78	99.63	99.78	99.63	99.78	99.63
	bless	98.90	98.59	98.88	98.62	98.88	98.61
	lighter	99.10	99.14	99.08	99.18	99.07	99.18
F-score	quake	94.51	65.56	94.54	65.61	94.51	65.57
	musket	96.06	95.68	96.05	95.69	96.05	95.68
	bless	98.79	97.94	98.78	98.04	98.77	98.04
	lighter	**99.26**	**98.58**	**99.22**	**99.06**	**99.23**	**99.09**
Gain	quake	89.58	48.76	89.64	48.82	89.59	48.78
	musket	92.40	91.70	92.39	91.71	92.39	91.69
	bless	97.58	95.90	97.57	96.11	97.54	96.09
	lighter	**98.52**	**97.17**	**98.44**	**98.12**	**98.46**	**98.18**

The α value for the 35×, 70×, 140× is 0.1, 0.05 and 0.025 respectively.

For Quake, we only measured the performance of the reported reads. Because Quake trimmed the untrusted tail of a read and throw away unfixable reads, it has a particularly high precision. But even if we only count the original errors in those reported reads, its sensitivity is very low. Lighter is among the best on different measurement and scenarios. This shows that our sampling method is able to distinguish most solid and weak kmer, and the error correction method is very effective.

We can show Lighter can achieve near constant space usage if the portion of set bits, namely occupancy rate, in the bloom filters stays almost the same. And this is true as shown in Table 1 given different coverages using simulated data set with 1% error rate. When the coverage is very low, the occupancy rate in table B is significant lower. This is because that when the coverage is very low, the count of the solid kmers is not

Coverage	Table A	Table B
10×	41.563	19.843
20×	41.555	32.765
35×	41.555	33.802
70×	41.580	33.906
140×	41.577	33.881
280×	41.571	33.903

Table 1: Occupancy rate(%) for each table for different coverages

very different from the weak kmers and the binomial test fails no matter how the α is set.

To further test the performance of Lighter, we showed how many kmers from the reference genome is stored in table B. We used the simulated data with 70x coverage and 1% error rate. There are totally 4,553,699 distinct kmers from the genome on one strand, and 4,553,653 of them are in table B which is almost all of them. We lose some kmers from the genome mostly due to the low coverage at the two ends of the chromosome.

The 6 simulated data sets are for haploid case, there are many species with a pair of chromosome in a cell, like human. Here we consider the case of diploid. We still use the e.coli genome as the reference genome and then introduce 0.1% SNPs to generate the second reference genome. Mason then will sample the same amount of reads from the two genomes, totally form an 70x coverage data set. There are 159,117 kmers containing the SNP position from one strand, and table B holds 158,987 of them. Though the performance is not as good as haploid case, it still hold almost all of them.

Effect of α

The sampling rate α is the key in Lighter which ensures the near constant space complexity. We did an evaluation on the simulated data set with 35× coverage and 1% error rate. The result is shown in Figure 3.

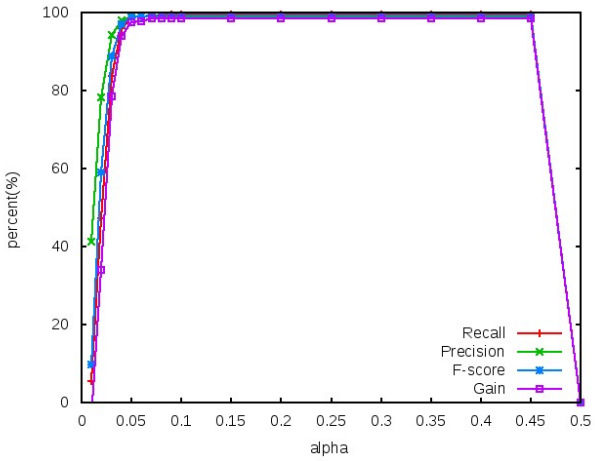

Figure 3: The effect of α

If α is too small, too much solid kmers will be failed to get in table A and too many positions will fail the hypothesis test. As a result, Lighter will try to correct those error-free region which also brings down the precision. And if α become even smaller, then there is no kmer in B, so Lighter will fail to correct all the reads because there is no candidate kmers.

If α is larger, y' will increase due to larger β and the $f(\alpha)$ becomes constant when $\alpha > 0.05$. As a result, it can handle this case nicely. However, when α is too large, the k-mer length k will be smaller than y', and all the positions will be regarded as untrusted. We can observe this from the dramatic drop in Figure 3.

This explanation can also be verified by looking at the occupancy rate of table A and B from the simulated data sets with 3 different coverage and 1% error rate when changing α shown in Figure 4. Also, we can see the occupancy rate of table A is almost the same given the same $\alpha \times$ coverage across the 3 data sets. And the occupancy rate of table B is very steady because the solid kmers from the genome are the same for the 3 data sets.

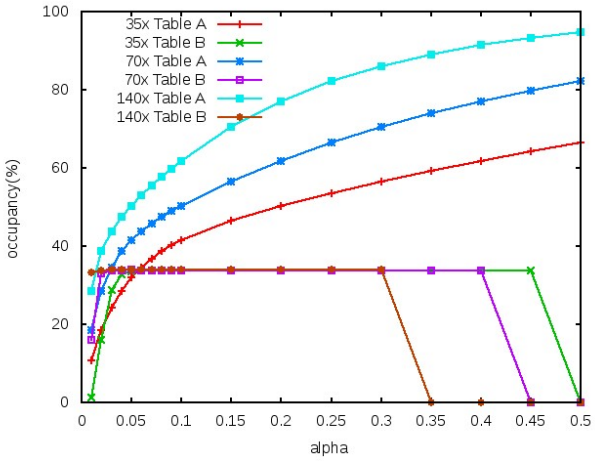

Figure 4: The effect of α on occupancy rate

Effect of kmer length

One important parameters for k-mer based error correction methods is the value of k. When k is too small, the k-mer containing sequencing error comes will likely be a k-mer from another location of the genome. When k is too large, there may not be enough error-free k-mers for us to do error correction, especially when the coverage is not high. From the perspective of efficiency, when k is large, we need more time to process a k-mer. And for the programs which requires storing the k-mer information, like using a hash table to store the counts, larger k causes more memory consumption. Since Lighter does not store the k-mers' information explicitly, larger k will only make Lighter slower but does not affect the memory usage. In [12], the authors suggested that we can select 4^k around two hundred times the genome size.

We measured the affect of k using the simulated data set with $35\times$ coverage and 1% error rate, the result is shown in Figure 5.

Real data set

E.Coli data set

We use ERR022075 data set which is a very deep coverage sequencing data set of ecoli K-12 strain genome. We still used Quake, Musket, Bless and Lighter to run this data set. Since usually we do not need that much of coverage, we uniformly sampled some reads from this data set and got a data set with roughly 75x coverage(total 3,497,616 reads). The reads in this data set is paired-end and the read length is 100 and 102. Because Bless can not handle paired-end reads with different length, we truncate the last 2 base to form a 100-length paired-end data set.

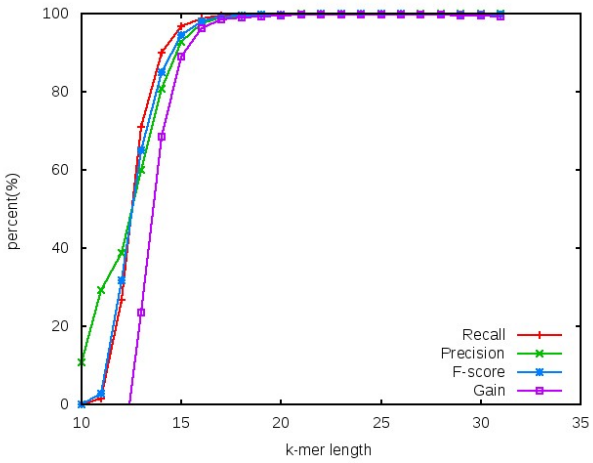

Figure 5: The effect of k-mer length

	Read Level			Base Level	
	Total Reads	Mapped Reads	Gain(%)	Match/Read	Gain(%)
Original	3,497,616	3,464,140	-	99.038	-
Quake	3,369,584	3,368,911	-2.75	97.305	-1.75
Musket	3,497,616	3,467,878	0.11	99.601	0.57
Bless	3,497,616	3,472,979	0.26	99.611	0.58
Lighter	3,497,616	3,476,425	0.35	99.611	0.58

Table 2: Alignment of E.Coli data set

Since the genome of E.Coli K-12 strain is fully sequenced, we measured that 4,519,538 out of 4,553,699 kmers from the genome is stored in table B, which is more than 99%. This number can be slightly improved using larger α.

We can also measure the error corrections' effect on the alignment. We use Bowtie2[cite] to align the reads to the reference genome and count the total number of the matched position.

In this table, "Gain" means how much improvement for its previous column comparing with the original data set. Match/Read represents the average matched position against the reference genome for the mapped reads.

For Quake, the gain is negative mainly due to unreported reads and trimmed 3' end. It turns out Lighter gives the most improvement for both read level and base level.

For those reads not got trimmed and alinged to the genome without indels (cigar field like 100M in the SAM file), we measure the matched ratio for each position shown in Figure 6. Since many of the reads in this data set start with N, the first base's matching ratio of the original data set is very low.

Another common measurement for error correction is to run the de novo assembly and see the impact to the contigs. We use Velvet[cite] to assemble the reads. Velvet is a de novo assembly software based on De Brujin graph. The parameter for Velvet especially the kmer size for the De Brujin graph is very important to the performance. We ran the Velvet on different parameters and choose the one with best N50 contig size for each program. Then for each chosen contig file, we evaluated the assembly quality using Quast[cite]. Since assemblers usually generates many very short contigs, we only choose the contigs whose length are no less than 100bp.

N50 is the length such that the total length of the contigs longer or equal to than this length covers half of the assembled genome. And NG50 is the length of the contig with respect to the reference genome. Edits

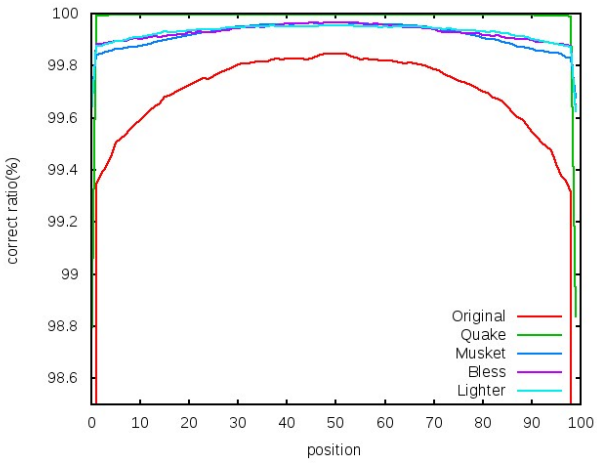

Figure 6: The matching ratio for each base in E.Coli data set

	N50	NG50	Edits / 100kbps	Misassemblies	Coverage(%)
Original	100,039	100,039	4.18	0	97.412
Quake	125,252	125,252	4.23	8	97.450
Musket	99,995	95,384	6.96	3	97.579
Bless	104,682	99,991	4.45	3	97.352
Lighter	105,574	104,682	5.02	5	97.548

Table 3: De novo assembly of E.Coli data set

per 100kbps is the number of mismatches or indels per 100kbps when aligning the contigs to the reference genome. Misassembly is a position of a contig that its left sequence and right sequence are mapped far away on the genome or overlap too much. The coverage are the portion of the reference genome that are covered by contigs.

Human Chr14

So far, we only tested the data on the genome of E.Coli, which is a pretty small. In this section, we use a data set from Gage[cite] on human chromosome 14. This data set is about 35x coverage and 101bp pair-end reads. And the k-mer size is 19 for all the programs in this data set.

Like for the E.Coli data set, we evaluate the result using both Bowtie2 and SOAPdenovo2.

The result of Bowtie2 is shown in Table 4.

	Read Level			Base Level	
	Total Reads	Mapped	Gain(%)	Match/Read	Gain(%)
Original	36,504,800	35,993,149	-	99.492	-
Quake	32,560,518	32,546,450	-9.58	93.411	-6.11
Musket	36,504,800	36,276,990	0.79	100.104	0.62
Bless	36,504,800	36,297,288	0.84	100.192	0.70
Lighter	36,504,800	36,280,350	0.80	100.109	0.62

Table 4: Alignment of chr14 data set

	N50	NG50	Edits / 100kbps	Misassemblies	Coverage(%)
Original	8096	5940	134.53	1097	78.490
Quake	7935	5809	143.91	2110	77.303
Musket	6261	4613	128.97	811	78.984
Bless	8397	6201	131.87	1425	78.747
Lighter	8370	6211	133.22	1416	78.858

Table 5: De novo assembly of chr14 data set

We also tested the effect to de novo assembly using Velvet and Quast which is shown in Table 5. In this data set, the improvement due to error corrections is obviously.

The results of the de novo assembly of the two real data set show that Lighter's result is comparable with other methods.

Running Time and Space Usage

The programs were run on a machine with 48 2.1GHz processors, 512G memory and 74T disk space.

Musket, Bless and Lighter are all aimming for low memory consumption. So we compared the peak memory usage as well as the peak disk consumption on the simulated data set with different coverage and with 1% error rate and the chr14 real data from Gage.

	35×		70×		140×		chr14	
	memory	disk	memory	disk	memory	disk	memory	disk
Quake	2.8G	3.3G	7.1G	6.0G	14G	12G	48G	57G
Musket	139M	0	160M	0	241M	0	1.9G	0
Bless	10M	661M	11M	1.3G	13M	2.6G	600M	15G
Lighter	31M	0	31M	0	31M	0	510M	0

From the table, we can see that Bless and Lighter can achieve nearly constant memory consumption as claimed. Musket took much less memory comparing with Quake but is much worse than Bless and Lighter.

For Bless, we increase the number of temporary files along with the coverage of the data set to achieve the constant memory consumption. Bless uses secondary memory to save the memory consumption, and the overall space consumption is much larger than Musket and Lighter. For the chr14 data set, The memory consumption of Bless can be decreased if we create more temporary files.

We compared the running time between Quake, Musket and Lighter using different number of threads on the simulated data set with 70× coverage and 1% error in Figure 7. Musket requires at least 2 threads due to its master-slave style parallelism, so its figure start at when number of threads is 2.

So far, Bless can only run in single-thread mode and it takes 1475s to finish which is much slower than Musket and Lighter.

The result shows that Lighter is very efficient and scalable.

Discussion

Lighter is a software tool for correcting sequencing errors in sequencing datasets. At Lighter's core is a method for obtaining a set of solid k-mers from a large collection of sequencing reads. Unlike previous methods, Lighter does this without counting k-mers in the reads. By setting its parameters appropriately, its memory usage and accuracy can be held almost constant with respect to depth of coverage. It is also quite fast and practical.

Though we demonstrate Lighter in the context of sequencing error correction, Lighter's counting-free approach could be applied in other situation where a collection of solid k-mers is desired. For example, one tool for scaling metagenome sequence assembly uses of a Bloom filter populated with solid k-mers as a memory-efficient, probabilistic representation of a De Bruijn graph [18]. Other tools use counting Bloom

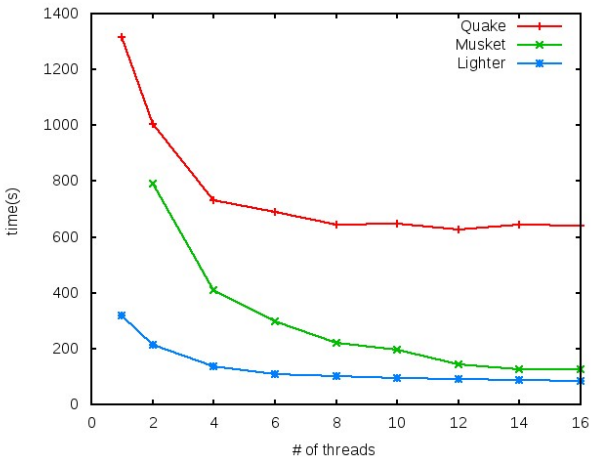

Figure 7: The running time on simulated data set with different number of threads

filters [2, 5] or the related CountMin sketch [4] to represent De Bruijn graphs for compression [10] or digital normalization and related tasks [25]. We expect Ideas from Lighter could be useful in reducing the memory footprint of these and other tools.

In this paper, we did not analyze the effect of α analytically and it is a difficult problem. So far, the size of table A and B are setting by considering unpractical pessimestic scenarios. If we know the effect in some compact form, we can optimize the performance of obtaining solid kmers by setting the best α and also use less space.

Lighter is free open source software released under the XXX license. The software and its source are available from https://github.com/mourisl/Lighter/.

Acknowledgements

Text for this section ...

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[3] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[6] T. C. Glenn. Field guide to next-generation dna sequencers. *Molecular Ecology Resources*, 11(5):759–769, 2011.

[7] E. C. Hayden. Is the $1,000 genome for real? *Nature News*, 2014.

[8] Y. Heo, X.-L. Wu, D. Chen, J. Ma, and W.-M. Hwu. Bless: Bloom-filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[9] L. Ilie, F. Fazayeli, and S. Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[10] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22):e171–e171, 2012.

[11] W.-C. Kao, A. H. Chan, and Y. S. Song. Echo: a reference-free short-read error correction algorithm. *Genome research*, 21(7):1181–1192, 2011.

[12] D. R. Kelley, M. C. Schatz, S. L. Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*, 11(11):R116, 2010.

[13] Y. Liu, J. Schröder, and B. Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

[14] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[15] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

[16] P. Melsted and B. V. Halldórsson. Kmerstream: Streaming algorithms for k-mer abundance estimation. *bioRxiv*, 2014.

[17] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):333, 2011.

[18] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[19] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[20] L. Salmela and J. Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[21] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, and B. Schmidt. Shrec: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163, 2009.

[22] H. Shi, B. Schmidt, W. Liu, and W. Müller-Wittig. A parallel algorithm for error correction in high-throughput short-read data on cuda-enabled graphics hardware. *Journal of Computational Biology*, 17(4):603–615, 2010.

[23] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1):131–155, 2012.

[24] X. Yang, K. S. Dorman, and S. Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533, 2010.

[25] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *arXiv preprint arXiv:1309.2975*, 2013.