Supplementary Materials: An Efficient Approach for Signature Profiling in Genomic Data through Variable-Length *k*-mers

Chelsea J.-T. Ju, Jyun-Yu Jiang, Ruirui Li, Zeyu Li, and Wei Wang

SI. Proof of Proposition 1

Proof. Given the prefix length i and the number of possible characters c, there are c^i possible prefixes in total. Assuming the characters are uniformly distributed, the probability that a particular prefix exists in n signatures is:

$$1 - \left(1 - \frac{1}{c^i}\right)^n.$$

Therefore, the expected number of collided prefixes is:

$$n - c^i\left(1 - \left(1 - \frac{1}{c^i}\right)^n\right),$$

and the expected number of prefixes without any collision is:

$$n - \left(n - c^i\left(1 - \left(1 - \frac{1}{c^i}\right)^n\right)\right) = c^i\left(1 - \left(1 - \frac{1}{c^i}\right)^n\right) = c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right).$$

However, the expected number above includes the cases that fail before reaching the i-th character. Hence, the expected number of these cases should be deducted from the above number. Finally, the expected number of signatures that fail to find their length-i prefixes along the trie during its insertion is:

$$c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right) - c^{i-1}\left(1 - \left(\frac{c^{i-1} - 1}{c^{i-1}}\right)^n\right).$$

$\square$

SII. Proof of Proposition 2

From Proposition 1, the expected number of node for prefix length i in n signatures is $c^i\left(1 - \left(\frac{c^i-1}{c^i}\right)^n\right)$. Intuitively, summing all possible prefix lengths up to the length of signature m, the expected number of trie nodes is $\sum_{i=1}^{m} c^i\left(1 - \left(\frac{c^i-1}{c^i}\right)^n\right)$. We provide the proper proof below.

Proof. Denote the expected number of signatures that fail to find their length-i prefixes on the trie during its insertion as $f(i)$. Given the length of signatures m, each signature that fails to find the length-i prefix along the trie during its insertion will result in the addition of $m - i + 1$ nodes. Based on Proposition 1, the expected number of nodes in the trie is:

$$\begin{aligned}
\sum_{i=1}^{m}(m-i+1)\cdot f(i) &= \sum_{i=1}^{m}(m-i+1)\left[c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right) - c^{i-1}\left(1 - \left(\frac{c^{i-1} - 1}{c^{i-1}}\right)^n\right)\right] \\
&= \sum_{i=1}^{m}(m-i+1)\left[c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right)\right] - \sum_{i=1}^{m-1}(m-i)\left[c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right)\right] \\
&= \sum_{i=1}^{m-1}\left[(m-i+1) - (m-i)\right]\left[c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right)\right] + c^m\left(1 - \left(\frac{c^m - 1}{c^m}\right)^n\right) \\
&= \sum_{i=1}^{m} c^i\left(1 - \left(\frac{c^i - 1}{c^i}\right)^n\right).
\end{aligned}$$

$\square$

SIII. Proof of Proposition 3

Proof. Suppose that the number of signatures to be added into a trie is extremely large. The expected number of nodes with c possible characters is:

$$\lim_{n\to\infty}\sum_{i=1}^{m} c^i\left(1-\left(\frac{c^i-1}{c^i}\right)^n\right)=\sum_{i=1}^{m}c^i=\frac{c\left(c^m-1\right)}{c-1}=\frac{c^{m+1}-c}{c-1}.$$

For the plain AC, there are four possible characters, i.e., `A`, `C`, `G` and `T`. Hence, the expected number of its nodes N_A is:

$$N_A=\frac{4^{m+1}-4}{4-1}=\frac{\left(2^2\right)^{m+1}-4}{3}=\frac{2^{2m+2}-4}{3}=\frac{4\left(2^{2m}-1\right)}{3}.$$

For the thinned automaton, there are two possible characters, i.e., `0` and `1`. Hence, the expected number of its nodes N_T is:

$$N_T=\frac{2^{m+1}-2}{2-1}=2(2^m-1).$$

Finally, we compute the ratio of the expected number of two approaches as follows:

$$\frac{N_T}{N_A}=\frac{2(2^m-1)}{\frac{4(2^{2m}-1)}{3}}=\frac{3}{2}\cdot\frac{2^m-1}{2^{2m}-1}=\frac{3}{2}\cdot\frac{2^m-1}{\left(2^m+1\right)\left(2^m-1\right)}=\frac{3}{2}\cdot\frac{1}{2^m+1}.$$

□

SIV. Related Works and Software Configurations

Existing *k*-mer counters index the reads into a compact and searchable structure, such as a hash table, a burst trie, or a compact suffix array. The occurrences of a specific *k*-mer can be retrieved by querying these data structures. Most of these counters are designed to process reads with fixed-size *k*-mers; several of them restrict the choice of *k* to fall within a threshold. These algorithms can be adapted to count *k*-mers of different sizes by repeating the process with different *k*'s. Here, we discuss these approaches.

A. Thread-Safe Shared Memory Hashing

Jellyfish [23] exploits the CAS (compare-and-swap) assembly instruction to update a memory location in a multi-threaded environment, and uses the "quotienting technique" and bit-packed data structure to reduce wasted memory. It provides a function (`--if=kmerfile`) to count only a list of specific *k*-mers with the same *k*. The counting step is repeated for different *k*. We use the following two command lines to perform counting:

```
$ jellyfish count -m k -t 1 -s 100M -C --if=kmerfile readfile -o countfile
$ jellyfish dump -c countfile >> countresult
```

Squeakr [27] builds an off-the-shelf data structure based on counting quotient filter (CQF). It maintains both global and local CQFs to facilitate updates of each thread. We first count the approximate occurrences of *k*-mers in reads, and then query the occurrences based on a list of specific *k*-mers. This process is repeated for different *k*. Note that the exact mode cannot handle small *k* where all *k*-mers appear in the data.

```
$ squeakr count -k k -s 20 -t 1 -o k_tmp readfile
$ squeakr query -f k_tmp -q kmerfile -o countfile
```

B. Disk-Based Hashing

Disk-based hashing reduces memory with complementary disk space. In general, this approach splits *k*-mers into bins, and stores them in files. Each bin is then loaded into the memory for counting.

DSK [4] divides *k*-mers using a specific hash function based on the targeted memory and disk space. To account for arbitrary *k*-mer lengths, we compile the source code of DSK based on the range of the *k*-mer sizes (`-DKSIZE_LIST=32`). For each experiment, we first load the *k*-mers into memory and determine the range of the *k*-mer sizes. We index reads with different *k*'s using the main program, and dump the result into a human readable format with `dsk2ascii`.

```
$ dsk -verbose 0 -file readfile -kmer-size k -abundance-min 0
    -out tmpdir/kmers.h5 -out-tmp tmpdir -out-compress 9
$ dsk2ascii -verbose -file tmpdir/kmers.h5 -out countfile
```

MSPKmerCounter (MSPKC) [21] proposes Minimum Substring Partitioning to reduce the memory usage of storing *k*-mers. Observing the fact that consecutive *k*-mers in a read often share a shorter substring, these consecutive *k*-mers can be compressed and stored in one bin. It is recommended to index reads with an odd number k less than 64. The software contains three functions to be run in sequence: `partition`, `count`, and `dump`. The partition step divides data using minimum substring partitioning, the count step computes the frequencies of existing *k*-mers, and the dump step converts the results into human readable format. This sequel is repeated for each k.

```
$ java -jar Partition.jar -in readfile -k k -L readlen -t 1
$ java -jar Count32.jar -k k -t 1
$ java -jar Dump64.jar -k k
```

KMC3 [18] scans reads one block at a time and uses a number of splitter threads to process the blocks. It leverages the concept of minimizer to further reduce disk usage. We use the main program from KMC3 to count the *k*-mer of all sizes seen in the list, with one k at a time. We set the `-cs` parameter to 4294967295 to ensure all of the frequently occurred *k-mers* are included.

```
$ kmc -t1 -sr1 -kK -ci0 -cs4294967295 -sf1 -sp1 readfile countfile tmpdir
```

C. *Probabilistic Hashing*

Probabilistic hashing approaches avoid counting the *k*-mers that are likely to arise from sequencing errors.
BFCounter [25] uses Bloom filter to identify all *k*-mers that are present more frequently than a threshold with a low false positive rate. The algorithm scans read data in two passes. The count function of BFCounter requires an estimation of the number of *k-mers*. We use *KmerStream* [24] to pre-compute the *k-mer* statistics in reads. We use the `dump` function to convert the results into human readable format to extract the frequencies.

```
$ KmerStream -k [list of ks] -o numK_file -t 1 readfile --tsv
$ BFCounter count -k k -n numK -t 1 -o kmers.compress readfile
$ BFCounter dump -k k -i kmers.compress -o countfile
```

khmer [35] uses a streaming-based probabilistic data structure, CountMin Sketch [11]. The algorithm is designed to perform in-memory nucleotide sequence *k*-mer counting, and cannot handle k larger than 32. We use its Python wrapper script, `load-into-counting`, to perform counting, which writes a *k*-mer graph for each k to files. We repeat this step for different k's. Each *k*-mer graph is loaded back to the memory one at a time, allowing us to query the count. We set the maximum amount of memory for data structure to be 16G as the required parameter.

```
$ khmerEnv/bin/load-into-counting.py -k k -M 16G -T 1 -q kmers.graph readfile
```

D. *Suffix-Arrays*

Suffix-arrays present the potential of searching arbitrary *k*-mers without any restriction of k on a single scan. However, constructing a suffix-array on read data can be computationally expensive.
Tallymer [19] is tailored to detect *de novo* repetitive elements ranging from 10 to 500bp in the genome. The algorithm first constructs an enhanced suffix array (`gt suffixerator`), and indexes *k*-mers one k at a time. We use `gt tallymer mkindex` to extract the *k*-mer index from the enhanced suffix-array, and use `gt tallymer search` to retrieve their counts.

```
$ gt suffixerator -dna -pl -tis -suf -lcp -v -parts 4 -db readfile -indexname tmp
$ gt tallymer mkindex -mersize k -minocc 1 -indexname kmer_index -counts -pl -esa tmp
$ gt tallymer search -output sequence counts -strand fp
    -tyr kmer_index -q kmerfile > countresult
```

MSBWT [15] compresses raw reads via a multi-string variant of Burrows-Wheeler Transform (BWT). Instead of concatenating all reads and sorting, it builds a BWT on each string and merges these multi-string BWTs through a small interleave array. The final structure allows a fast query of *k*-mers of arbitrary k. To index short reads, we use

```
$ msbwt cffq --uniform --compressed -p 1 tmpdir readfile
```

; to index long reads, we use

```
$ msbwt cffq -p 1 tmpdir readfile
```

. The following command line is used for querying:

```
$ msbwt massquery --rev-comp tmpdir kmerfile
```

E. *Burst Tries*

KCMBT [22] uses a cache efficient burst trie to store compact *k*-mers. The trie structure stores *k*-mers that share the same prefix in the same container. When a container is full, *k*-mers are sorted and burst. A good balance between the container size and the tree depth is essential to avoid constant sorting and bursting. As a result, KCMBT uses hundreds of trees. Unfortunately, it is limited to process *k*-mers with k less than 32. We first load the list of *k*-mers into memory. KCMBT generates binary files containing *k-mers* and their counts. We use `kcmbt_dump` to convert the binary data into human readable files.

```
$ kcmbt -k k -i readfile -t 1
$ kcmbt_dump 1
```

SV. Scalability Analysis

A. *The Effect of Reads and K-mers on Single Thread*

We evaluate the scalability of different approaches using the synthetic datasets. On the read level, we use 1.2 million *k*-mers ranging from 15-151bp (*wide*) to evaluate the effect of different read lengths and number of reads. On the *k*-mer level, we use 86,976,737 reads of 180bp to evaluate the effects of different batches of *k*-mers.

Results are shown in Table S1 and S2. The total run-time and memory of each approach are highlighted in boldface font. The run-time is further broken down into the automaton construction phase (Prep) and the read querying phase (Query) for TahcoRoll and PlainAC_Py. The two phases of MSBWT and KMC3 include indexing the reads (Prep) and querying the *k*-mers (Query). Read processing is performed in the querying phase of TahcoRoll and PlainAC_Py, but in the preparation phase of MSBWT and KMC3. Therefore, the run-time of querying is not on the same scale across different approaches. For Jellyfish, we use its function to count the list of *k*-mers directly, so the run-time cannot be split in details. Dagger (†) marks the most time efficient approach; asterisk (∗) marks the most memory efficient approach.

TABLE S1: Time (Hour) and memory (GB) of profiling 1.2 million *k*-mers of 15-151bp on different read sets.

Read Length	**Total Reads**	**TahcoRoll**				**PlainAC_Py**				**MSBWT**				**KMC3**				**Jellyfish**	
		Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	**Time**	**Mem**
75bp	10,128,312	0.006	0.09	**0.10**†	**3.29**∗	0.02	0.11	**0.13**	**6.75**	0.40	0.01	**0.41**	**5.86**	1.49	0.02	**1.51**	**3.30**	**1.83**	**4.74**
	34,497,448	0.005	0.28	**0.29**†	**3.29**∗	0.02	0.37	**0.39**	**6.75**	0.90	0.01	**0.91**	**7.88**	2.45	0.09	**2.53**	**5.52**	**5.55**	**4.74**
	97,011,938	0.005	0.78	**0.78**†	**3.29**∗	0.02	1.04	**1.06**	**6.75**	3.26	0.02	**1.95**	**17.57**	5.82	0.68	**6.50**	**8.00**	**15.24**	**4.74**
100bp	11,397,007	0.005	0.13	**0.14**†	**3.29**∗	0.03	0.17	**0.20**	**6.75**	0.45	0.01	**0.46**	**6.40**	2.42	0.33	**2.75**	**3.83**	**3.32**	**4.74**
	41,054,662	0.005	0.47	**0.48**†	**3.29**∗	0.02	0.59	**0.61**	**6.75**	1.29	0.01	**1.30**	**11.10**	4.81	0.61	**5.42**	**7.56**	**11.25**	**4.74**
	114,813,452	0.006	1.35	**1.36**†	**3.29**∗	0.02	1.59	**1.61**	**6.75**	3.49	0.02	**3.51**	**26.00**	16.23	3.35	**19.58**	**20.10**	**31.78**	**4.74**
125bp	10,822,319	0.004	0.15	**0.15**†	**3.29**∗	0.03	0.19	**0.22**	**6.75**	0.63	0.01	**0.65**	**6.83**	2.81	0.81	**3.61**	**4.15**	**5.26**	**4.74**
	58,012,701	0.005	0.77	**0.78**†	**3.29**∗	0.03	0.99	**1.02**	**6.75**	2.59	0.02	**2.61**	**17.48**	10.53	2.51	**13.04**	**19.03**	**27.22**	**4.74**
	107,375,244	0.005	1.37	**1.38**†	**3.29**∗	0.02	1.84	**1.87**	**6.75**	4.56	0.02	**4.58**	**29.75**	18.46	3.92	**22.37**	**34.59**	**50.41**	**4.74**
150bp	27,628,054	0.006	0.35	**0.36**†	**3.29**∗	0.02	0.55	**0.57**	**6.75**	1.69	0.01	**1.71**	**11.46**	9.09	1.88	**10.97**	**14.87**	**18.78**	**4.74**
	57,437,772	0.007	1.20	**1.21**†	**3.29**∗	0.02	1.20	**1.22**	**6.75**	3.50	0.02	**3.51**	**20.31**	17.26	3.98	**21.24**	**31.10**	**36.86**	**4.74**
	114,306,300	0.006	2.01	**2.01**†	**3.29**∗	0.03	2.42	**2.44**	**6.75**	5.86	0.02	**5.88**	**37.27**	33.45	8.25	**41.69**	**58.23**	**74.29**	**4.74**
180bp	16,197,631	0.006	0.35	**0.35**†	**3.29**∗	0.03	0.40	**0.43**	**6.75**	2.43	0.01	**2.45**	**9.30**	7.45	1.99	**9.44**	**14.61**	**15.51**	**4.74**
	37,836,905	0.005	0.86	**0.87**†	**3.29**∗	0.02	0.87	**0.90**	**6.75**	3.20	0.02	**3.22**	**16.96**	16.05	4.20	**20.26**	**33.34**	**35.14**	**4.74**

TABLE S2: Time (Hour) and memory (GB) of profiling different *k*-mer sets on 86,976,737 synthetic reads of 180bp.

K-mer Batch	**Total K-mers**	**TahcoRoll**				**PlainAC_Py**				**MSBWT**				**KMC3**				**Jellyfish**	
		Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	Prep	Query	**Time**	**Mem**	**Time**	**Mem**
Small (15-31bp)	1,200,000	0.0004	2.56	**2.56**	**0.51**∗	0.003	1.81	**1.81**†	**1.25**	5.39	0.01	**5.40**	**34.35**	3.99	0.35	**4.34**	**14.61**	**11.15**	**0.83**
	6,000,000	0.002	4.83	**4.83**	**2.09**	0.02	2.42	**2.44**†	**5.70**	5.39	0.06	**5.46**	**34.31**	5.39	0.84	**6.23**	**14.61**	**11.11**	**0.83**∗
	12,000,000	0.003	5.48	**5.48**	**3.85**	0.03	2.74	**2.77**†	**10.93**	5.39	0.12	**5.51**	**34.35**	5.89	0.95	**6.84**	**14.61**	**11.17**	**0.83**∗
	24,000,000	0.006	7.22	**7.23**	**7.13**	0.09	3.11	**3.21**†	**20.93**	5.39	0.23	**5.63**	**34.35**	5.94	0.96	**6.91**	**14.61**	**11.15**	**0.83**∗
Medium (65-81bp)	1,200,000	0.005	2.01	**2.01**†	**2.82**	0.03	2.42	**2.45**	**5.83**	5.39	0.01	**5.40**	**34.35**	5.03	2.59	**7.62**	**58.16**	**11.41**	**2.47**∗
	6,000,000	0.02	2.47	**2.49**†	**13.49**	0.13	4.77	**4.90**	**28.59**	5.39	0.06	**5.45**	**34.35**	4.90	1.91	**6.81**	**58.16**	**11.37**	**2.47**∗
	12,000,000	0.09	3.53	**3.62**†	**26.52**	0.27	5.27	**5.54**	**56.71**	5.39	0.11	**5.50**	**34.33**	4.90	1.93	**6.83**	**58.16**	**11.07**	**2.47**∗
	24,000,000	0.16	4.00	**4.16**†	**52.11**	0.75	5.25	**6.00**	**112.5**	5.39	0.22	**5.61**	**34.35**	4.87	1.49	**6.37**	**58.16**	**11.36**	**2.47**∗
Large (131-151bp)	1,200,000	0.02	2.65	**2.67**†	**6.10**	0.06	2.98	**3.04**	**12.24**	5.39	0.02	**5.41**	**34.35**	3.51	2.35	**5.87**	**67.27**	**6.66**	**4.74**∗
	6,000,000	0.08	3.51	**3.59**†	**29.91**	0.29	4.34	**4.63**	**60.63**	5.39	0.08	**5.47**	**34.35**	4.28	4.04	**8.32**	**67.27**	**6.72**	**4.74**∗
	12,000,000	0.18	4.37	**4.55**†	**58.43**	0.55	4.98	**5.53**	**118.97**	5.39	0.16	**5.55**	**34.33**	5.14	4.50	**9.65**	**69.19**	**8.59**	**4.38**∗
	24,000,000	0.42	4.42	**4.84**†	**117.79**	1.33	4.90	**6.23**	**240.67**	5.39	0.29	**5.69**	**34.35**	4.10	3.05	**7.17**	**67.27**	**6.73**	**4.74**∗

B. Improvement with Multiple-Threads

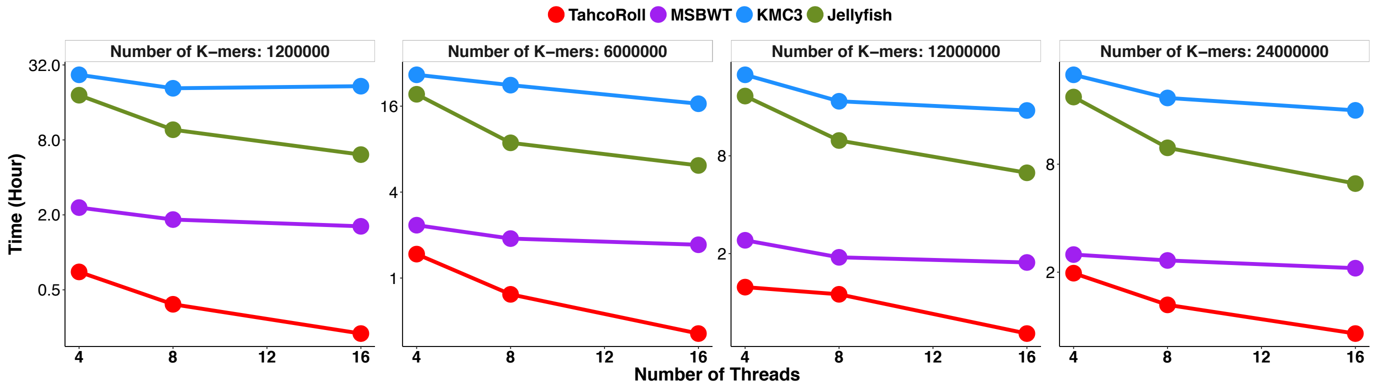

Fig. S1: Multi-threads evaluation of MSBWT, KMC3, Jellyfish and TahcoRoll while profiling *wide* batches of *k*-mers on 86,976,737 synthetic reads of 180bp.

TABLE S3: Time (Hour) and memory (GB) of profiling different *wide* batch of *k*-mers on 86,976,737 synthetic reads of 180bp. Dagger(†) marks the most time efficient approach; asterisk (*) marks the most memory efficient approach.

Total K-mers	Methods	4-Thread (Hour)	8-Thread (Hour)	16-Thread (Hour)	Memory (GB)
1,200,000	**TahcoRoll**	0.70†	0.38†	0.22†	3.50*
	MSBWT	2.29	1.83	1.62	30.83
	Jellyfish	18.29	9.66	6.10	4.74
	KMC3	26.68	20.79	21.61	72.83
6,000,000	**TahcoRoll**	1.47†	0.77†	0.41†	16.07
	MSBWT	2.35	1.89	1.71	30.83
	Jellyfish	19.39	8.86	6.15	4.74*
	KMC3	26.55	22.47	16.65	72.77
12,000,000	**TahcoRoll**	1.24†	1.12†	0.64†	31.49
	MSBWT	2.42	1.90	1.76	30.83
	Jellyfish	18.66	9.94	6.29	4.74*
	KMC3	25.22	17.33	15.21	73.08
24,000,000	**TahcoRoll**	1.98†	1.32†	0.91†	61.86
	MSBWT	2.51	2.33	2.11	30.83
	Jellyfish	18.96	9.88	6.24	4.74*
	KMC3	22.22	18.74	15.97	73.08

SVI. Evaluation of Binarized Representations

TABLE S4: Evaluation of different binarized representations. Time is reported in hour and memory is reported in gigabyte. Nucleotides can be divided into balanced or unbalanced partitions. The *p*-values are computed through paired *t*-tests on time against the default setting: [{`A`, `G`}, {`C`, `T`}], and adjusted by Bonferroni correction.

Mapping	**0={A,C}; 1={G,T}**		**0={A,T}; 1={C,G}**		**0={A}; 1={C,G,T}**		**0={C}; 1={A,G,T}**		**0={G}; 1 ={A,C,T}**		**0={T}; 1={A,C,G}**	
	Time	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**
SRR1293902	1.28	4.49	1.27	4.42	1.54	3.36	1.42	3.20	1.54	3.12	1.93	2.98
SRR1293901	1.35	4.49	1.39	4.42	1.72	3.36	1.71	3.20	1.79	3.12	1.96	2.98
GSM1254204	1.26	4.49	1.26	4.42	1.46	3.36	1.64	3.20	1.70	3.12	2.01	2.98
SRR5951587	0.32	11.11	0.30	10.84	0.34	9.85	0.47	7.93	0.57	7.85	0.54	9.27
SRR5951588	0.52	11.11	0.48	10.84	0.58	9.85	0.83	7.93	1.21	7.85	0.81	9.27
SRR5951600	0.76	11.11	0.74	10.84	0.78	9.85	1.46	7.93	1.36	7.85	1.11	9.27
***p*-value**	0.3408		0.12714		0.033552		0.043938		0.008988		0.010212	