Med. Rev. 2021; 1(2): 114–125 DE GRUYTER

Research Article

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

TahcoRoll: fast genomic signature profiling via thinned automaton and rolling hash

https://doi.org/10.1515/mr-2021-0016
Received July 5, 2021; accepted November 11, 2021;
published online February 14, 2022

Abstract

Objectives: Genomic signatures like *k*-mers have become one of the most prominent approaches to describe genomic data. As a result, myriad real-world applications, such as the construction of de Bruijn graphs in genome assembly, have been benefited by recognizing genomic signatures. In other words, an efficient approach of genomic signature profiling is an essential need for tackling high-throughput sequencing reads. However, most of the existing approaches only recognize fixed-size *k*-mers while many research studies have shown the importance of considering variable-length *k*-mers.
Methods: In this paper, we present a novel genomic signature profiling approach, TahcoRoll, by extending the Aho–Corasick algorithm (AC) for the task of profiling variable-length *k*-mers. We first group nucleotides into two clusters and represent each cluster with a bit. The rolling hash technique is further utilized to encode signatures and read patterns for efficient matching.
Results: In extensive experiments, TahcoRoll significantly outperforms the most state-of-the-art *k*-mer counters and has the capability of processing reads across different sequencing platforms on a budget desktop computer.
Conclusions: The single-thread version of TahcoRoll is as efficient as the eight-thread version of the state-of-the-art, JellyFish, while the eight-thread TahcoRoll outperforms the eight-thread JellyFish by at least four times.

Keywords: Aho–Corasick algorithm; genome sequencing; k-mers; multiple pattern matching; rolling hash.

Chelsea J.-T. Ju and Jyun-Yu Jiang contributed equally to this work.

***Corresponding author: Wei Wang**, Department of Computer Science, University of California, Los Angeles, USA,
E-mail: weiwang@cs.ucla.edu. https://orcid.org/0000-0002-8180-2886
Chelsea J.-T. Ju, Jyun-Yu Jiang, Ruirui Li and Zeyu Li, Department of Computer Science, University of California, Los Angeles, USA

Introduction

Genomic signature profiling is a popular approach to decode important information from sequencing data. These genomic signatures called *k*-mers are short consecutive substrings of a genomic sequence and represent certain signatures to characterize different genomes or different regions in one genome. Instead of alignment, existing lightweight approaches pre-compute a searchable database of *k*-mers representing the genomic signatures, and count the occurrences of these signatures in sequencing data. RNA and metagenomic sequencing are the predominant fields that use *k*-mer approaches. To name a few methods, Sailfish [1], RNA-Skim [2], and Kallisto [3] are prevalent for RNA-Seq transcript quantification; LMAT [4] and Kraken [5] present efficient strategies to assign taxonomic labels for each metagenomic read. QAPA [6] leverages *k*-mer sequences for conducting the systematic analysis of alternative polyadenylation from RNA sequences. Minimap2 [7] maps sequences against a large reference database by finding primary chains and indexing homopolymer compressed *k*-mers. Other *k*-mer applications include studying the CpG evolution in mammalian genomes using *k*-mer and *k*-flank patterns [8], comparing *k*-mer profiles of family trios to detect disease-causing variants [9], and mining differentially occurred *k*-mers between cases and controls for association mapping [10].

Most of the existing applications employ a set of fixed-size *k*-mers; however, selecting the appropriate *k* is challenging. If a *k*-mer is too long, it can fail to map a read with sequencing errors. On the other hand, if a *k*-mer is too short, it can appear everywhere in the read data. In addition, the best *k* to characterize different genomic regions can vary.

Several genome assemblers, such as SPAdes [11], Velve [12], SOAPdenovo [13], tringTie [14], TransBorrow [15], PIM-Assembler [16], ALGA [17], MAC [18], and Raven [19], recognize the impact of *k*-mer sizes and consider building the de Bruijn graph with different sizes of *k*-mers. Chae et al. [8] have shown that it is necessary to consider patterns of 3–10-mers to construct the phylogenetic tree. Rahman

Open Access. © 2021 Chelsea J.-T. Ju et al., published by De Gruyter. BY-NC-ND This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

et al. [10] have proposed to merge the differentially occurred *k*-mers to form longer sequences, resulting in variable-length sequences, for downstream analysis. Ju et al. [20] have also demonstrated the advantage of using variable-length *k*-mers for transcript abundance quantification and splice junction prediction. To the best of our knowledge, there are two existing approaches capable of generating variable-length *k*-mers as a set of signatures of interests. One exploits the suffix-tree structure to discover the shortest uncommon substrings [20], which represent the signatures of different transcript sequences. The other employs a pattern-growth approach to generate frequent *k*-mers of variable sizes, *vl*-mers [21], for both DNA and amino acid sequences. Despite the efforts in discovering variable-length *k*-mers in DNA sequences, current *k*-mer counters are optimized to process *k*-mers of a fixed length. Inevitably, it is critical to have a feasible data structure to store *k*-mers of different sizes, and to analyze sequencing data efficiently and accurately.

Given a set of *k*-mers with the same size, a straightforward counting approach is to index the given *k*-mers with a hash table, and examine through read sequences with a fixed-size window. If a set includes *k*-mers of different sizes, the read sequences require to be scanned multiple times with different *k*'s. This repetition limits the analysis to assess only a small range of k's. An alternative method is to use existing efficient *k*-mer counting algorithms. Reads are first indexed and stored as *k*-mers, and the number of occurrences of each *k*-mer can be computed from the index. One of these counters, Jellyfish [22], has been widely used as the underlying structure for Sailfish, Kraken, and DIAMUND [23]. Jellyfish employs the thread-safe approach to efficiently measure the frequencies of the *k*-mers. Techniques used by other counters include disk-based hashing, probabilistic hashing, lock-free chaining hashing, suffix-array structure, and burst tries. Several of these implementations, such as khmer [24], KCMBT [25], KmerEstimate [26], and CHTKC [27], restrict *k* to fall below a threshold to mitigate the memory usage and run time.

The suffix-array-based approach is the only one that offers the potential to deal with *k*-mers of variable-lengths. All other approaches are meant to tackle sequences with a fixed *k*. Thus, repeat the counting for different *k*'s is unavoidable.

Measuring the frequencies of a set of *k*-mers with different sizes can be reduced to a multiple pattern matching problem [28] in computer science. A linear solution to examine the reads once is the Aho–Corasick algorithm (AC) [29], which creates a tree automaton upon the trie of keywords. In this trie, there are additional links between internal nodes to facilitate the k-mer matching without backtracking, i.e., jumping back and forth of the query sequence. A drawback of maintaining this automaton is the memory requirement for storing long or myriad *k*-mers. As we increase the number or the length of *k*-mers, the tree grows wider and deeper respectively, which produces more nodes and links to facilitate the traversal. In addition, larger *k*-mers are usually more diverse and have shorter common prefixes, requiring more space for *k*-mer representation. Fortunately, the concise representation of DNA molecules allows further reduction in memory requirement of this automaton. Since these k-mers are composed of only four different characters: A, C, G, and T, they can be succinctly represented in a binary format. Traditionally, each nucleotide is encoded into two bits for its binary representation. Here, we propose to use an even more concise representation with one bit. We partition these four characters into two groups, and use one bit, i.e., 0 or 1, to represent them. This binarized representation allows us to significantly shrink the structure of the trie, and to substantially reduce the memory. The degenerated representation can cause collisions where different k-mers are encoded with identical binarized representation. To avoid this collision, each node on the tree contains a hash table to facilitate recovering the original *k*-mers.

It is noteworthy to mention that our goal is not to compute the frequencies of all possible *k*-mers with different sizes, but to profile a pre-defined set of variable-length *k*-mers as signatures in sequencing reads. The focus of this paper is also different from assembling reads with variable sizes of *k*-mers. An example of a pre-defined set can carry the genetic markers of different microorganisms in metagenomic research. Since the term signatures here refer to a set of representative *k*-mers, we use these two terminologies interchangeably throughout the paper. Our contributions are three-fold. First, we highlight the need of having a viable data structure to store and to profile *k*-mers of different sizes in DNA sequences. Second, to the best of our knowledge, this is the first study to profile a vast amount of pre-defined set of variable-length *k*-mers simultaneously in genomic data. We propose to apply the AC with a memory efficient automaton. Third, we leverage the properties of DNA sequences to construct an efficient in-memory structure and employ the rolling hash technique to accelerate the match. We adapt existing k-mer counters to perform the same task, and conduct a comprehensive analysis over 13 different methods. Results show that our method, TahcoRoll, is more efficient in profiling signatures with a wide range of sizes than conventional k-mer counters. It is also resistant to the change of read length and quantity. The parallelization of TahcoRoll has demonstrated a promising improvement over different numbers of threads, where the parallelizations of KMCs and MSBWT are constrained by the disk I/O. Most importantly, TahcoRoll can investigate reads from Illumina, PacBio, and Oxford Nanopore on a commodity desktop computer while KMC3 and MSBWT fail on long reads.

Related work

Thread-safe shared memory hashing

Jellyfish [22] exploits the 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 also provides a function to count only a list of specific k-mers. **Squeakr** [30] 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. **CHTKC** [27] constructs a lock-free chaining hash table for multi-thread hash accesses.

Disk-based hashing

Disk-based hashing reduces memory usage with complementary disk space. In general, this method breaks k-mers into bins, and reserves them in files. Each bin is then placed into the memory for calculation. **DSK** [31] splits k-mers into bins using a specific hash function based on the targeted memory and disk space. **MSPKmerCounter (MSPKC)** [32] presents a novel technique, Minimum Substring Partitioning, to lower the memory consumption of storing k-mers. Recognizing 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 suggested to index reads with an odd number k less than 64. **KMC** [33], **KMC2** [34], and **KMC3** [35] are serial developments of parallel counters. These methods scan reads one block at a time, and utilize several splitter threads to tackle these blocks. KMC2 leverages the concept of minimizer to further reduce disk usage. KMC3 speeds up the running time and optimizes the memory usage by taking a larger part of input data and better balancing the bin sizes.

Probabilistic hashing

To avoid calculating the counts of k-mers with sequencing errors, BFCounter [36] uses Bloom filter to identify all k-mers that are present more frequently than a threshold with a low false-positive rate. The algorithm examines read data in two passes to avoid reporting false-positive counts. **khmer** [24] uses a streaming-based probabilistic data structure, CountMin Sketch [11]. The algorithm is designed to conduct in-memory counting, and cannot tackle k larger than 32.

Suffix-arrays

Suffix-arrays show the potential of examining 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** [37] is tailored to detect *de novo* repetitive elements ranging from 10 to 500 bp in the genome. The algorithm first constructs an enhanced suffix-array, and indexes k-mers for a fixed value of k. The indexing step needs to be repeated for different k's.

MSBWT [38] is designed to consolidate raw reads via a multi-string variant of Burrows–Wheeler Transform (BWT). Instead of concatenating all reads and sorting, MSBWT establishes a BWT on each string and merges these multi-string BWTs through a small interleave array. The final structure supports a fast query of k-mers of arbitrary k.

Burst tries

An obvious shortcoming of tree-based and hash-based solutions is that they can suffer from enormous cache misses when working on massive datasets. **KCMBT** [25] uses a cache efficient burst trie to store compact k-mers. The trie stores k-mers that share the same prefix in the same container. When a container is full, k-mers are sorted and burst. An appropriate balance between the container size and the tree depth is necessary to prevent constant sorting and bursting. As a result, KCMBT requires to employ hundreds of trees.

Unfortunately, it is limited to handle k-mers with k less than 32.

Methods

In this section, we present the thinned Aho–Corasick automaton accelerated by rolling hash (TahcoRoll) to profile variable-length k-mers in genomic data.

Problem statement

We focus on counting the occurrences of a set of representative k-mers instead of all possible variable-length k-mers because of the following two reasons. First, the number of all variable-length k-mers in a DNA sequence is huge. More specifically, the lower bound of time complexity to investigate occurrences of all possible k-mers with different k is at least $O(L^2)$ for a read length L. Second, some existing works [20, 21] focus on discovering relevant variable-length k-mers for various applications. Given a list of signatures, it is not necessary to profile all possible k-mers.

Signature p	Sequencing Reads T AATTGAGAT	ATTGACATCG	Number of Occurrences c_p
ATT			2
GA			3
TTG			2
AGAT			1
TC			1

Figure 1: An example with five signatures and two sequencing reads in signature profiling. Each segment represents an occurrence of the corresponding signature in the read.

Suppose that $\boldsymbol{P}$ is the set of representative k-mers as signatures, where the length k is different across the set. Given a set of sequencing reads $\boldsymbol{T}$, our goal is to profile the occurrences of each signature $p \in \boldsymbol{P}$. Note that the lengths of occurred patterns are shorter than the read length.

More formally, for each signature $p \in \boldsymbol{P}$, we aim to develop an efficient algorithm to compute the number of overall occurrences c_p is:

$$c_p = \sum_{t \in \boldsymbol{T}} |\{i | t[i \ldots i + |p| - 1] = p, 1 \le i \le |t| - |p| + 1\}|,$$

where $|p|$ and $|t|$ indicate the lengths of the signature p and the sequencing read t, and $t[i \ldots j]$ denotes the substring of t from the i-th to the j-th character. For each signature $p \in \boldsymbol{P}$, occurrences in the read set $\boldsymbol{T}$ are counted as a number c_p, which is the objective of signature profiling. Figure 1 shows an example with five signatures $\boldsymbol{P} = \{\mathrm{ATT}, \mathrm{GA}, \mathrm{TTG}, \mathrm{AGAT}, \mathrm{TC}\}$ and two sequencing reads $\boldsymbol{T} = \{\mathrm{AATTGAGAT}, \mathrm{ATTGACATCG}\}$.

Framework overview: Figure 2 illustrates the overall framework of our proposed TahcoRoll that consists of the automaton construction phase for pre-processing and the read query phase for profiling. In the automaton construction phase, given a signature set for profiling, we first binarize original signatures into binarized signatures, thereby building the corresponding thinned automaton in the manner of Aho–Corasick automaton. In the read query phase, we traverse the thinned automaton and leverage the rolling hash technique to accelerate the verification process for profiling.

Aho–Corasick automaton

To address signature profiling, an intuitive way is to reduce the task into multiple pattern matching [28] by mapping signatures onto patterns and each set of reads onto the input text. Multiple pattern matching algorithms find all occurrences in a read for each signature, so the profiling results can be obtained by aggregating these occurrences. We propose to apply the AC [29], one of the state-of-the-art approaches for multiple pattern matching, to profile signatures.

AC conducts the matching process along a trie that corresponds to patterns. Each node in AC has a failure link that allows fast transitions from one node to the other representing its longest possible suffix without backtracking. Informally, AC constructs a finite state machine (or an automaton) that resembles a trie and failure links. The pattern matching process can be treated as transitions between nodes in the automaton, and failure links provide efficient transitions between failed matches. Figure 3 shows an example of AC with five signatures. For example, the node of signature ACAT has a failure link to the node of AT. Given a sequencing read ATTTC to be profiled, AC will first match the signature ATT in the *blue* node. Then, it fails to match the third T and transits to the *orange* node that still has no child of T. After traveling along the failure link again to the *yellow* node, both the last two characters TC can proceed towards the *orange* and *brown* nodes that indicate a match of signature TTC.

The construction of the automaton in AC with signatures $p \in \boldsymbol{P}$ only requires a simple breadth-first search (BFS) with a linear time complexity $O(\sum_{p \in \boldsymbol{P}} |p|)$. To profile signatures in reads $O(t \in \boldsymbol{T})$, AC only needs to simulate transitions on the automaton, which also has a linear time complexity $O(\sum_{t \in \boldsymbol{T}} |t| + \sum_{p \in \boldsymbol{P}} c_p)$. The space complexity of AC is also linear, $O(\sum_{p \in \boldsymbol{P}} |p|)$, to maintain a node and a constant of links for each character. In theory, AC is a perfect fit for signature profiling.

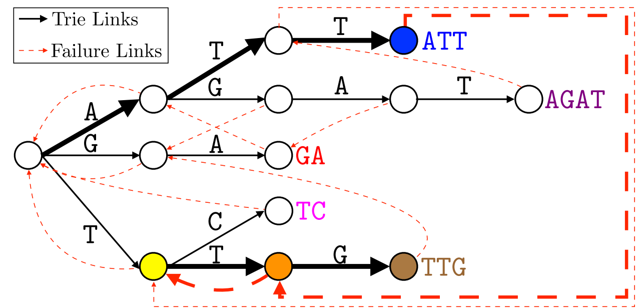

Figure 3: The automaton of AC with five signatures. Black solid links are trie links, and red dashed links are failure links. Colored nodes and thicker links are traversed while profiling the read ATTTG.

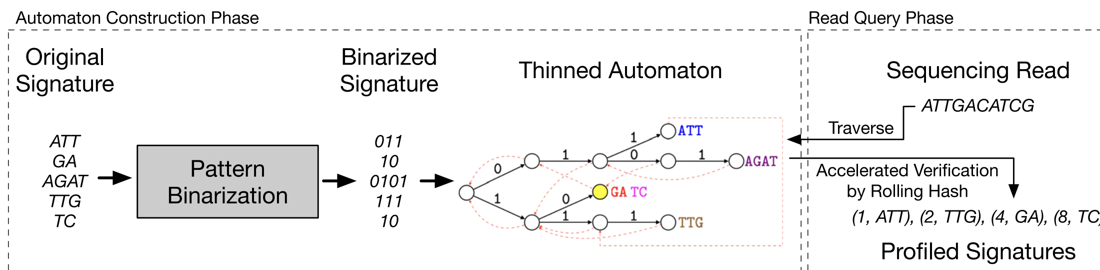

Figure 2: An example with five signatures and two sequencing reads in signature profiling. Each segment represents an occurrence of the corresponding signature in the read.

Thinned automaton with binarized pattern matching

Even though we have shown the theoretical capability of AC for signature profiling, there are still some hurdles for AC in practice. One of the most critical issues is the memory usage when the number of signatures is huge. More specifically, each character in signatures can be referred to as a trie node, which provides plenty information and consumes a considerable amount of memory. For example, the Python implementation of AC requires more than 240 GB of memory to process 24 million signatures whose lengths range from 135 to 151. Especially for signatures with fewer and shorter common prefixes, nodes tend to have more child nodes. The greater width leads to an increase in memory usage.

To reduce both the number of nodes and the width of the automaton, we propose the thinned automaton with binarized pattern matching. More formally, each signature $p[1 \dots |p|] \in \boldsymbol{P}$ is transformed into a binarized pattern $p'[1 \dots |p|]$ before being added into the automaton. The i-th character $p'[i]$ of p' is defined as follows:

$$p'[i] = \text{binarize}(p[i]),\ \ \text{where}\ \text{binarize}(c) = \begin{cases} 0, c \in \{\text{A}, \text{G}\} \\ 1, c \in \{\text{C}, \text{T}\} \end{cases}.$$

Note that these four characters can be randomly divided into two groups. In practice, we suggest grouping characters so that the constructed trie could be as balanced as possible based on data distribution. Besides, if the data distribution is too complicated to have a suitable grouping method for characters, a limitation of our proposed method is the inevitable effect on the efficiency. In this paper, we use a balanced partition which groups A and G together. Compressing two characters into one bit 0 or 1, binarized patterns improve the representation capability of a depth-d node in the trie from 1 to 2^d unbinarized pattern(s), thereby reducing both the width of the automaton and the number of nodes. In this paper, the automaton with binarized patterns is named *thinned automaton* because of its reduced width. Here, we conduct a theoretical analysis of the improvement of the thinned automaton against the plain AC. For convenience, we assume that each character in a signature is uniformly distributed. To estimate the worst-case scenario, we assume that every signature has the largest length m observed in the set. While inserting a signature into a trie, the number of newly added nodes depends on the presence of its prefixes in the trie. Proposition 1 gives an expectation of finding prefixes for n signatures with c possible characters.

Proposition 1 (Proved in Section S1 in the Supplementary Material) *Given n signatures with c possible characters to be added into a trie, the expected number of signatures that fail to find their length-i prefixes along the trie during its insertion is* $c^i\left(1 - \left(\frac{c^i-1}{c^i}\right)^n\right) - c^{i-1}\left(1 - \left(\frac{c^{i-1}-1}{c^{i-1}}\right)^n\right)$, *where* $0 \le i \le m$.

Based on Proposition 1, we derive the expected number of nodes in a trie in Proposition 2.

Proposition 2 (Proved in Section S2 in the Supplementary Material) *Given n signatures of length m with c possible characters to be added into a trie, the expected number of trie nodes is* $\sum_{i=1}^{m}\left[c^i - c^i\left(\frac{c^i-1}{c^i}\right)^n\right]$.

	Signature p	Sequencing Reads $\boldsymbol{T}$
Original	ATT GA TTG AGAT TC	AATTGAGAT ATTGACATCG
Binarized	011 10 111 0101 10	001110101 0111000101

Figure 4: The binarized representations for five patterns and two sequencing reads. Two signatures GA and TC share the same binarized pattern (red). A substring in a sequencing read ATCG has the identical binarized form to the signature AGAT (blue).

Following Proposition 2, Proposition 3 derives the expected improvement on the number of trie nodes when the number of signatures is approaching to a large number.

Proposition 3 (Proved in Section S3 in the Supplementary Material) *When the number of signatures in the automaton is approaching to a large number, the expected number of nodes in the thinned automaton is only* $\frac{3}{2} \cdot \frac{1}{2^m+1}$ *of those in the plain AC.*

As shown in Proposition 3, the improvement with the thinned automaton is guaranteed under the assumption mentioned above. However, DNA sequences are biased. In this scenario, where the characters of each signature are not uniformly distributed, the improvement can be more pronounced because more duplicated segments lead to fewer trie nodes.

Even though the thinned automaton reduces the number of nodes, compressed representations may lead to collisions. Figure 4 shows an example of binarized results for five patterns and two sequencing reads, and Figure 5 further illustrates the corresponding thinned automaton. Two signatures CA and TG share the same binarized pattern 10 and result in a collision when reaching the *yellow* node in Figure 5. Substrings with identical binarized representations may also lead to false matches. For instance, ATGC in the second read, which is not a signature, has the same binarized representation 0101 as the signature ACAT.

To maintain the correctness of signature profiling, each match to a binarized pattern needs to be verified with the original signatures. In other words, it is very time-consuming if there are serious collisions in certain nodes. A naïve comparison costs $O\left(\sum_{p\in\{p|p'=h,p\in\boldsymbol{P}\}} |p|\right)$ time to verify signatures with the same representation h.

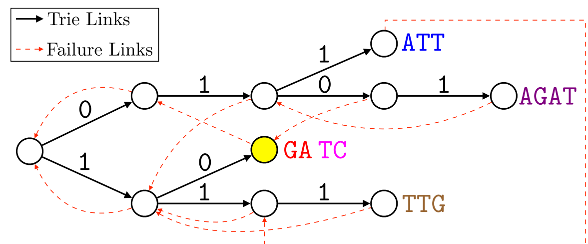

Figure 5: An example of the thinned automaton of AC with five signatures. Black solid links are trie links, and red dashed links are failure links. The *yellow* node represents two signatures GA and TC.

Acceleration by rolling hash

Using hash functions is an intuitive idea to speed up comparisons between strings. As the lengths of signatures vary, arbitrary substrings of the read $t \in \boldsymbol{T}$ is required to compute hash values during verification. However, on-the-fly computation of hash values takes an additional linear time $O(|t|)$ for each checkup; pre-computing all possible substrings is also infeasible due to dispensable computations and extensive $O(|t|^2)$ additional memory.

To accelerate verification, we propose to apply rolling hash [39] that alleviates the time complexity for each checkup from linear to constant with a linear time pre-processing and additional linear memory consumption. Rolling hash is a family of hash functions where the input is hashed with a window that moves through the input. A new hash value can be rapidly calculated from the given old hash value in $O(1)$ time. % time complexity. It also allows $O(1)$ query time on the hash value of any substring in the input with content-based slicing. We implement the Rabin-Karp algorithm [40] as the rolling hash function. Formally, the hash value of a length-L input $t[1 \dots L]$ is defined as follows:

$$H(t[1 \dots L]) = t[1]a^{L-1} + \cdots + t[L-1]a^1 + t[L]a^0 \pmod q,$$

where $t[i]$ is the i-th character of the input; a is a constant multiplier; q is a constant prime modulus. The hash value of a length-i prefix of t can be recursively calculated through the hash value of the length-$(i-1)$ prefix:

$$H(t[1 \dots i]) = \begin{cases} H(t[1 \dots i-1]) \cdot a + t[i], & \text{if } i > 1 \\ t[1], & \text{if } i = 1 \end{cases} \pmod q.$$

With bottom-up computation, hash values of all prefixes $H(t[1 \dots i])$ can be preprocessed in both $O(\mathrm{L})$ time and space complexity. Given the hash values of all prefixes, the hash value of a substring $t[i \dots j]$ can be derived in $O(1)$ as follows:

$$H(t[i \dots i]) = \begin{cases} H(t[1 \dots j]) - H(t[1 \dots i-1]) \cdot a^{j-i+1}, & \text{if } i > 1 \\ H(1 \dots j), & \text{if } i = 1 \end{cases} \pmod q.$$

As a theoretical analysis, Proposition 4 gives a theoretical upper-bound of the collision probability. The larger the prime modulus q, the smaller the hash collision probabilities. Note that we employ the Rabin-Karp algorithm instead of cyclic polynomials for hashing because the former method has been demonstrated to be more efficient for general applications [41].

Proposition 4 (Gonnet and Baeza-Yates [42]**)** *The probability of two different random strings of the same length having the same hash value in Rabin-Karp rolling hash is $P(collision) \leq \frac{1}{q}$, where q is the prime modulus in computations of the Rabin-Karp algorithm.*

To apply rolling hash for acceleration, each node contains a hash table that maps a hash value onto the original signature. When transitioning to the node, the hash value of the matching substring in the read can be rapidly calculated and verified for its presence in the hash table.

As a result, the average time complexity of each checkup reduces to $O(1)$. The overall time complexity of TahcoRoll is $O(\sum_{p \in \boldsymbol{P}} |p| + \sum_{t \in \boldsymbol{T}} |t| + \sum_{p \in \boldsymbol{P}} c_p)$, including the construction of the automaton and the matching process. The only memory overhead is hash tables with exactly $|\boldsymbol{P}|$ values, which is an amortized $O(|\boldsymbol{P}|)$ space.

Results

Experimental datatsets

The performance of different algorithms is affected by four factors: signature lengths, number of signatures, read length, and number of reads. The flexibility of synthetic reads allows us to closely examine the effects of these factors. The randomly generated signatures are designed to test the worst scenario as their characters are uniformly distributed and may not share as many common prefixes as in the real sequencing applications. We also generate signatures from both genomic and transcriptomic sequences to analyze real reads from a diverse range of sequencing platforms. Synthetic datasets are available at *https://figshare.com/s/6f02feaf89c4ff6ddc9e*.

Synthetic signatures

To examine the effects of signature number and length, we generate four batches of k-mers with different lengths, denoted by *small* (15–31 bp), *medium* (65–81 bp), *large* (131–151 bp), and *wide* (15–131 bp). Each batch contains four sets of 1.2, 6, 12, and 24 million k-mers. These numbers are arbitrarily chosen to examine the scalability of different methods. The sequence of each k-mer is randomly assigned with four nucleotide characters and a random length that falls in the appropriate range. Each k-mer is represented by its canonical form.

Synthetic reads

We used polyester [43] to generate 15 sets of RNA-Seq experiments, with read lengths of 75, 100, 125, 150, and 180 bp. Each set contains 10–115 million reads from randomly selected transcripts based on Ensembl Human Genome RCh38 [44].

Real datasets

The first dataset contains two experiments to study the transcriptomic analyses for lymphoblastoid cells [23]: SRR1293901 is a 2×262 cycle run from Illumina MiSeq and SRR1293902 is a 2×76 cycle run from Illumina HiSeq 2000. The second dataset, GSM1254204, aims to characterize the transcriptome of human embryonic stem cells using PacBio long reads [45]. The third set is generated by Oxford Nanopore to study the whole genome of breast cancer model cell line with different read lengths: SRR5951587, SRR5951588, and SRR5951600. For the RNA-Seq datasets, we use a list of 10,962,469 k-mers selected from transcript

sequences that can distinguish different transcript isoforms. For the WGS datasets, 10,935,397 short sequences are randomly selected from the reference genome as signatures. Since long reads contain a higher error rate, we cannot set the *k*-mer size too long.

Implementation details

TahcoRoll builds a thinned automaton on a set of signatures represented by their canonical form (i.e., the lexicographical minimum of itself and its reverse complementary sequence).

Each node of the automaton holds an unordered map for average constant time complexity of searches and insertions after querying binarized patterns on the structure. The memory consumption of all automaton nodes is simultaneously pre-allocated for efficient memory operations. This is achieved by estimating the number of nodes through binary searches on sorted patterns. The profiling process scans each read twice, one for its forward sequence and the other for its reverse complementary sequence. The operations of the rolling hash are optimized by pre-computing the powers of the prime modulus.

In addition, the paralleled version applies the multi-threading capability of C++14. % for implementation. Unless otherwise mentioned, most of the experiments in this paper are conducted on a server with 504 GB of memory and two Intel Xeon E5-2680 v2 @ 2.80 GHz CPUs, where each CPU offers 10 cores.

Software adaption

We include all the *k*-mer counters mentioned in Section "Introduction" for experiments.

We also implement two baseline methods. The first one is a naïve implementation in C++, denoted by "Naïve". It uses a hash table to store *k*-mers and scans through the reads multiple times with different window sizes. Theoretically, Naïve is light in memory, but requires an extensive running time. The second baseline is the conventional AC. We test two publicly available implementations written in Python (PlainAC_Py; pyahocorasick 1.1.3) and C++ (PlainAC_C++; cjgdev/aho_corasick).

Automaton construction

The memory of AC is sensitive to the composition of signature patterns such as *k*-mer lengths, the number of *k*-mers, and common prefixes shared by different *k*-mers. Figure 6 compares the computational resources used for automaton construction in TahcoRoll against PlainAC_Py and PlainAC_C++ over 16 sets of signatures. The implementation of PlainAC_C++ uses several additional data structures on each node to facilitate the traversal on an automaton, causing a huge memory overhead. As a result, PlainAC_C++ is fast in automaton construction but requires twice and five times more memory than PlainAC_Py and TahcoRoll, respectively. For the *large* batch of 24 million *k*-mers, PlainAC_C++ maxes out the memory capacity (>396 GB) of our server, and thus the recorded run-time is truncated. Our thinned automaton consistently requires less time than PlainAC_Py in construction. As we increase the number of *k*-mers, the construction time rises. The memory of the thinned automaton is significantly reducing to nearly half of the memory required in PlainAC_Py.

Pilot study of 13 approaches

We perform a preliminary assessment of the memory footprint and run-time on 11 existing counters, together with Plain_AC and TahcoRoll.

We separate the analyses into two panels as demonstrated in Figure 7. The top panel focuses on different numbers of reads, and the bottom panel focuses on

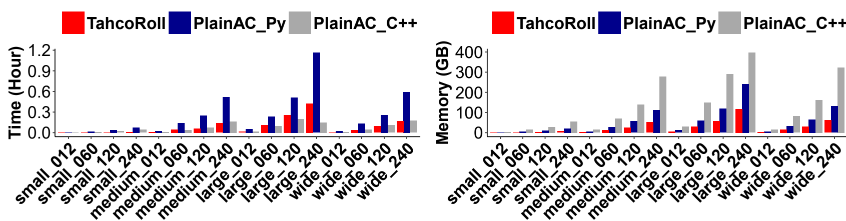

Figure 6: Run-time and memory for constructing the automaton given 16 sets of *k*-mer patterns. A lower value represents a more efficient approach. PlainAC$_C++ maxes out the memory capacity while constructing the "large" batch of 24 million *k*-mers (large_240), and thus its recorded time and memory are truncated. TahcoRoll consistently requires less time and memory than PlainAC_Py.

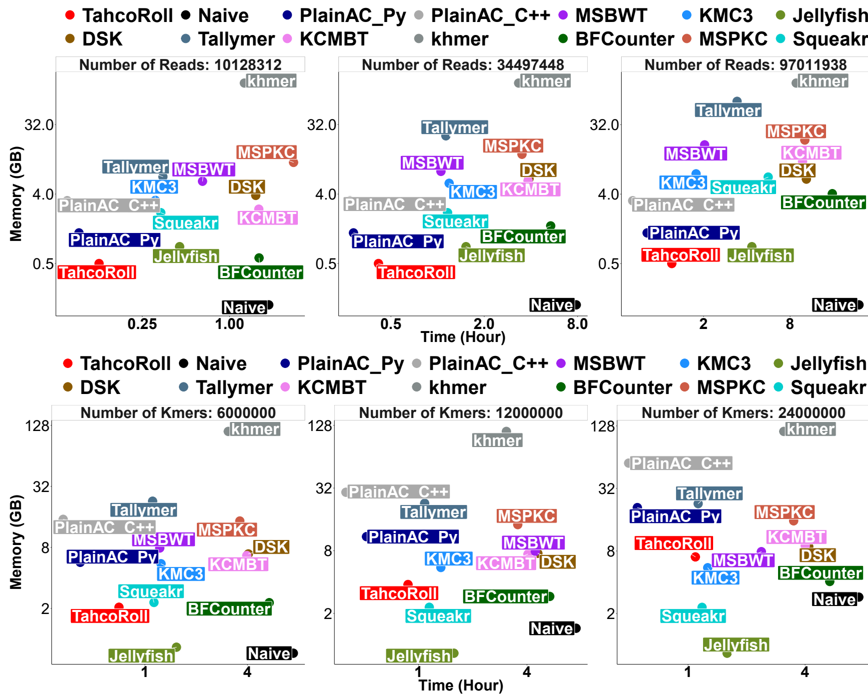

Figure 7: Run-time (x-axis) and memory (y-axis) of counting *small* batches *k*-mers on synthetic reads of 75 bp. Each point represents a pair of measurements (run-time and memory). Points on the lower left corner of each plot indicate more efficient approaches. The top panel examines different read sets with 1.2 million *k*-mers; the bottom panel examines different *k*-mer sets with 34,497,448 reads.

different numbers of *k*-mers. Methods in the bottom-left corner of each plot indicate being both time and memory efficient. As we predicted, Naïve uses very little memory, but takes a long time to complete. PlainAC is fast but requires a large amount of memory when increasing the number of *k*-mers. Consistent with the analysis in Figure 6, PlainAC_C++ uses twice as much memory as PlainAC_Py.

TahcoRoll is the most efficient approach in five out of these six analyses. KMC3 and Squeakr use less memory when there are 24 million *k*-mers, but requires more time than TahcoRoll. When we fix the number of *k*-mers (top panel), the memory and run-time for KMC3 and Squeakr increase with the number of reads, but the memory stays constant for both TahcoRoll and Jellyfish. Jellyfish is memory efficient when counting a given list of *k*-mers with the same size; however, repeating this process for different *k*'s makes it more time-consuming than TahcoRoll.

Extensive study on synthetic datasets with both single and multiple threads

We use 1.2 million *k*-mers ranging from 15–151 bp (*wide*) to evaluate the scalability on different read lengths and number of reads. We highlight the total run-time and memory consumption of each approach in Table 1. 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. Its memory usage depends on the size of the list of *k*-mers, and can be as efficient as TahcoRoll. However, its run-time does not scale well with datasets containing more or longer reads. TahcoRoll consistently outperforms others on different read sets in run-time and memory.

Next, we use 86,976,737 reads of 180 bp to evaluate the scalability on different batches of *k*-mers, which are designed to test the worst scenario. Table 2 shows that when the *k*-mers are short (*small*), PlainAC_Py uses the least amount of time. When *k*-mers get longer, TahcoRoll is the most efficient approach. This observation is due to less collision in the signature sets. Under a severe condition where there is a large number (12 and 24 million) of *k*-mers with uniformly distributed characters, TahcoRoll requires more memory than MSBWT in three out of six cases. It is worth mentioning that MSBWT and KMC3 write a huge amount of intermediate files to disk (at least 16 GB for MSBWT and 43 GB for KMC3 for this dataset) to alleviate

Table 1: Time (h) and memory (GB) of synthetic signatures over different read sets.

Read length	Total reads	TahcoRoll				PlainAC_Py				NISBWT				KMC3				Jellyfish	
		Prep	Query	Time	Mem	Prep	Query	Time	Mem	Prep	Query	Time	Mem	Prep	Query	Time	Mem	Time	Mem
75 bp	10,128,312	0.006	0.09	**0.10**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**
100 bp	11,397,007	0.005	0.13	**0.14**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**
125 bp	10,822,319	0.004	0.15	**0.15**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**
150 bp	27,628,054	0.006	0.35	**0.36**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**
180 bp	16,197,631	0.006	0.35	**0.35**[a]	**3.29**[b]	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**[a]	**3.29**[b]	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**

[a]Marks the most time efficient approach. [b]Marks the most memory efficient approach. Time and Memory consumption are more important than the remaining columns and are therefore in bold.

Table 2: Time (h) and memory (GB) of synthetic reads over different *k*-mer sets.

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 (15–31 bp)	1,200,000	0.0004	2.56	**2.56**	**0.51**[b]	0.003	1.81	**1.81**[a]	**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**[a]	**5.70**	5.39	0.06	**5.46**	**34.31**	5.39	0.84	**6.23**	**14.61**	**11.11**	**0.83**[b]
	12,000,000	0.003	5.48	**5.48**	**3.85**	0.03	2.79	**2.77**[a]	**10.93**	5.39	0.12	**5.51**	**34.35**	5.89	0.95	**6.84**	**14.61**	**11.17**	**0.83**[b]
	24,000,000	0.006	7.22	**7.23**	**7.13**	0.09	3.11	**3.21**[a]	**20.93**	5.39	0.23	**5.63**	**34.35**	5.94	0.96	**6.91**	**14.61**	**11.15**	**0.83**[b]
Medium (65–81 bp)	1,200,000	0.005	2.01	**2.01**[a]	**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**[b]
	6,000,000	0.02	2.47	**2.49**[a]	**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**[b]
	12,000,000	0.09	3.53	**3.62**[a]	**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**[b]
	24,000,000	0.16	4.00	**4.16**[a]	**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**[b]
Large (131–151 bp)	1,200,000	0.02	2.65	**2.67**[a]	**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**[b]
	6,000,000	0.08	3.51	**3.59**[a]	**29.91**	0.29	4.39	**4.63**	**60.63**	5.39	0.08	**5.47**	**34.35**	4.28	4.09	**8.32**	**67.27**	**6.72**	**4.74**[b]
	12,000,000	0.18	4.37	**4.55**[a]	**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**[b]
	24,000,000	0.42	4.42	**4.84**[a]	**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]

[a]Marks the most time efficient approach. [b]Marks the most memory efficient approach. Time and Memory consumption are more important than the remaining columns and are therefore in bold.

Table 3: Time (h) and memory (GB) of profiling synthetic reads over *wide* batches *k*-mers.

Total *k*-mers	Methods	Four-thread	Eight-thread	16-thread	Memory
1,200,000	TahcoRoll	0.70[a]	0.38[a]	0.22[a]	3.50[b]
	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[a]	0.77[a]	0.41[a]	16.07
	MSBWT	2.35	1.89	1.71	30.83
	Jellyfish	19.39	8.86	6.15	4.74[b]
	KMC3	26.55	22.47	16.65	72.77
12,000,000	TahcoRoll	1.24[a]	1.12[a]	0.64[a]	31.49
	MSBWT	2.42	1.90	1.76	30.83
	Jellyfish	18.66	9.94	6.29	4.74[b]
	KMC3	25.22	17.33	15.21	73.08
24,000,000	TahcoRoll	1.98[a]	1.32[a]	0.91[a]	61.86
	MSBWT	2.51	2.33	2.11	30.83
	Jellyfish	18.96	9.88	6.24	4.74[b]
	KMC3	22.22	18.74	15.97	73.08

[a]Marks the most time efficient approach. [b]Marks the most memory efficient approach.

the memory bottleneck. In contrast, TahcoRoll is an in-memory approach that does not generate any intermediate data.

MSBWT, KMC3, and Jellyfish allow indexing reads in parallel, so we evaluate the parallel settings on the *wide* batches of *k*-mers. Table 3 shows the run-time and memory usage of analyzing 86,976,737 synthetic reads of 180 bp across four sets of *k*-mers. Both Jellyfish and TahcoRoll scale well with the number of threads, but the improvement of MSBWT and KMC3 is marginal. This is mainly due to the limitation of I/O as these two approaches constantly read and write files to disk. The run-time of TahcoRoll remains faster than others across different experiments and threads. The four-thread TahcoRoll demonstrates to be faster than others with 16 threads.

Data from different sequencing platforms

Here, we examine the practical usage by analyzing signatures from real DNA sequences with reads from different sequencing platforms. Table 4 summarizes the nature and analysis of each dataset. TahcoRoll is run with a single-thread and eight-threads; others are run with eight-threads. For the measurement that is less efficient than TahcoRoll, we compute the fold-change to those reported by the eight-thread TahcoRoll. MSBWT is unable to finish indexing for the PacBio data within two days. KMC3 cannot index long reads from Nanopore as the data exceeds the buffer size set by the program; it also consumed all memory available on the machine (32G) for PacBio data. Overall, the run-time of single-thread TahcoRoll is as efficient as Jellyfish with eight-threads and significantly outperforms KMC3 in short reads and MSBWT in long reads. In parallel settings, TahcoRoll runs at least four times faster than MSBWT and Jellyfish and demonstrates a drastic improvement over KMC3.

Conclusions

In this paper, we present a novel challenge of variable-length *k*-mer profiling in genomic sequences. While the

Table 4: Evaluation of datasets from different sequencing platforms.

Dataset		SRR1293902	SRR1293901	GSM1254204	SRR5951587	SRR5951588	SRR5951600
Source		RNA-Seq	RNA-Seq	RNA-Seq	WGS	WGS	WGS
Platform		Illumina HiSeq	Illumina MiSeq	PacBio	Nanopore	Nanopore	Nanopore
Number of reads		38,278,052	9,524,186	3,239,918	205,685	171,398	161,148
Average read length		75	262	1,113	3 kb	8 kb	12 kb
Number of *Sig-mers*		10,962,469	10,962,469	10,962,469	10,935,397	10,935,397	10,935,397
Lengths of *Sig-mers*		25–60	25–60	25–60	25–60	25–60	25–60
Time (h)	TahcoRoll (1-thread)	1.20	1.40	1.17	0.27	0.44	0.65
	TahcoRoll (8-thread)	0.23	0.28	0.22	0.06	0.09	0.16
	MSBWT	0.95 (4.1X)	1.64 (5.8X)	*NA*	3.03 (53.4X)	2.79 (29.5X)	12.31 (77.7X)
	KMC3	15.85 (68.5X)	14.16 (50.7X)	19.26 (87.3X)		*exceed buffer size*	
	Jellyfish	0.94 (4.0X)	1.56 (5.6X)	1.13 (5.1X)	0.27 (4.8X)	0.48 (5.1X)	I 0.68 (4.2X)
Memory (GB)	TahcoRoll	4.18	4.17	4.18	10.5	10.5	10.5
	MSBWT	7.76 (1.8X)	70.10 (16.8X)	NA	1.89	3.16	4.65
	KMC3	28.76 (6.8X)	24.84 (5.9X)	31.34 (7.4X)		*exceed buffer size*	
	Jellyfish	1.79	1.79	1.79	1.79	1.79	1.79

MSBWT, KMC3, and Jellyfish are run with eight-threads. Fold-change is relative to those reported by eight-thread TahcoRoll.

necessity of diversifying *k*-mer lengths has already been shown in many studies [8, 20, 21], most existing studies only allow fixed-length *k*-mers and need a significant amount of memory, disk space, and time to profile *k*-mers with a wide range of *k*'s. By leveraging the techniques of binarization and rolling hash for Aho–Corasick automaton, we construct an in-memory approach to profile variable-length *k*-mers in genomic data without the requirement of any disk space.

A pilot study provides a comprehensive overview of the strengths and limitations of 13 approaches. Additional experiments demonstrate that TahcoRoll scales well with both longer and more reads, especially that its memory usage is independent of the read data. It is the only approach that can efficiently process data from different sequencing platforms.

Although our experiments focus on counting the occurrences of a set of *k*-mers, the thinned automaton can be expanded to store essential information for each *k*-mer, such as its explicit positions in a genome. TahcoRoll opens up the opportunity to profile a set of variable-length *k*-mers, especially for long reads. It can be used as a standalone software package or to be integrated into existing pipelines for transcript quantification and microbial community profiling.

Research funding: The work was partially funded by NSF DGE-1829071, NIH R35-HL135772, NIH/NIBIB R01-EB027650. The funding organizations played no role in the study design; in the collection, analysis, and interpretation of data; in the writing of the report; or in the decision to submit the report for publication.

Author contributions: Chelsea Ju and Jyun-Yu Jiang equally contributed to the concept and design of the proposed method, implementation, experiments, and paper writing. Ruirui Li and Zeyu Li implemented several baseline methods and participated in discussions for analysis. Wei Wang supervised the project.

Competing interests: None.

Informed consent: Not applicable.

Ethical approval: Not applicable.

Further statements: TahcoRoll is open-source ad may be downloaded from https://github.com/chelseaju/TahcoRoll.

References

1. Patro R, Mount SM, Kingsford C. Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. Nat Biotechnol 2014;32:462–4.
2. Zhang Z, Wang W. RNA-Skim: a rapid method for RNA-Seq quantification at transcript level. Bioinformatics 2014;30: i283–92.
3. Bray NL, Pimentel H, Melsted P, Pachter L. Near-optimal probabilistic RNA-seq quantification. Nat Biotechnol 2016;34: 525–7.
4. Ames SK, Hysom DA, Gardner SN, Lloyd GS, Gokhale MB, Allen JE. Scalable metagenomic taxonomy classification using a reference genome database. Bioinformatics 2013;29:2253–60.
5. Wood DE, Salzberg SL. Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biol 2014;15:1–2.
6. Ha KC, Blencowe BJ, Morris Q. QAPA: a new method for the systematic analysis of alternative polyadenylation from RNA-seq data. Genome Biol 2018;19:1–8.
7. Li H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics 2018;34:3094–100.
8. Chae H, Park J, Lee SW, Nephew KP, Kim S. Comparative analysis using K-mer and K-flank patterns provides evidence for CpG island sequence evolution in mammalian genomes. Nucleic Acids Res 2013;41:4783–91.
9. Salzberg SL, Pertea M, Fahrner JA, Sobreira N. DIAMUND: direct comparison of genomes to detect mutations. Hum Mutat 2014; 35:283–8.
10. Rahman A, Hallgrímsdóttir I, Eisen M, Pachter L. Association mapping from sequencing reads using k-mers. Elife 2018;7: e32920.
11. Bankevich A, Nurk S, Antipov D, Gurevich AA, Dvorkin M, Kulikov AS, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J Comput Biol 2012;19: 455–77.
12. Zerbino DR, Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res 2008;18: 821–9.
13. Xie Y, Wu G, Tang J, Luo R, Patterson J, Liu S, et al. SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads. Bioinformatics 2014;30:1660–6.
14. Kovaka S, Zimin AV, Pertea GM, Razaghi R, Salzberg SL, Pertea M. Transcriptome assembly from long-read RNA-seq alignments with StringTie2. Genome Biol 2019;20:1–3.
15. Yu T, Mu Z, Fang Z, Liu X, Gao X, Liu J. TransBorrow: genome-guided transcriptome assembly by borrowing assemblies from different assemblers. Genome Res 2020;30:1181–90.
16. Angizi S, Fahmi NA, Zhang W, Fan D, PIM-Assembler. A processing-in-memory platform for genome assembly. In: 2020 57th ACM/IEEE design automation conference (DAC). IEEE; 2020: 1–6 pp.
17. Swat S, Laskowski A, Badura J, Frohmberg W, Wojciechowski P, Swiercz A, et al. Genome-scale de novo assembly using ALGA. Bioinformatics 2021;37:1644–51.
18. Tang L, Li M, Wu FX, Pan Y, Wang J. MAC: merging assemblies by using adjacency algebraic model and classification. Front Genet 2020;10:1396.
19. Vaser R, Sikic M. Raven: a de novo genome assembler for long reads. BioRxiv 2021:2020–08.
20. Ju CJ, Li R, Wu Z, Jiang JY, Yang Z, Wang W. Fleximer: accurate quantification of RNA-Seq via variable-length k-mers. In: Proceedings of the 8th ACM international conference on bioinformatics, computational biology, and health informatics. ACM; 2017:263–72 pp.

21. Zhang J, Guo J, Yu X, Yu X, Guo W, Zeng T, et al. Mining k-mers of various lengths in biological sequences. In: International symposium on bioinformatics research and applications. Cham: Springer; 2017:186–95 pp.
22. Marçais G, Kingsford C. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics 2011; 27:764–70.
23. Cho H, Davis J, Li X, Smith KS, Battle A, Montgomery SB. High-resolution transcriptome analysis with long-read RNA sequencing. PLoS One 2014;9:e108095.
24. Zhang Q, Pell J, Canino-Koning R, Howe AC, Brown CT. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. PLoS One 2014;9: e101271.
25. Mamun AA, Pal S, Rajasekaran S. KCMBT: ak-mer counter based on multiple burst trees. Bioinformatics 2016;32:2783–90.
26. Behera S, Gayen S, Deogun JS, Vinodchandran NV. KmerEstimate: a streaming algorithm for estimating k-mer counts with optimal space usage. In: Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics. ACM; 2018:438–47 pp.
27. Wang J, Chen S, Dong L, Wang G. CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table. Briefings Bioinf 2021;22:bbaa063.
28. Navarro G, Raffinot M. Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences. Cambridge University Press; 2002.
29. Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM 1975;18:333–40.
30. Pandey P, Bender MA, Johnson R, Patro R. Squeakr: an exact and approximate k-mer counting system. Bioinformatics 2018;34: 568–75.
31. Rizk G, Lavenier D, Chikhi R. DSK: k-mer counting with very low memory usage. Bioinformatics 2013;29:652–3.
32. Li Y. MSPKmerCounter: a fast and memory efficient approach for k-mer counting. arXiv preprint arXiv:1505.06550 2015.
33. Deorowicz S, Debudaj-Grabysz A, Grabowski S. Disk-based k-mer counting on a PC. BMC Bioinf 2013;14:1–2.
34. Deorowicz S, Kokot M, Grabowski S, Debudaj-Grabysz A. KMC 2: fast and resource-frugal k-mer counting. Bioinformatics 2015;31: 1569–76.
35. Kokot M, Długosz M, Deorowicz S. KMC 3: counting and manipulating k-mer statistics. Bioinformatics 2017;33:2759–61.
36. Melsted P, Halldórsson BV. KmerStream: streaming algorithms for k-mer abundance estimation. Bioinformatics 2014;30: 3541–7.
37. Kurtz S, Narechania A, Stein JC, Ware D. A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. BMC Genom 2008;9:1–8.
38. Holt J, McMillan L. Merging of multi-string BWTs with applications. Bioinformatics 2014;30:3524–31.
39. Cohen JD. Recursive hashing functions for n-grams. ACM Trans Inf Syst 1997;15:291–320.
40. Karp R. Efficient randomized pattern-matching algorithms. IBM J Res Dev 1987:31:249–60.
41. Lemire D, Kaser O. Recursive n-gram hashing is pairwise independent, at best. Comput Speech Lang 2010;24:698–710.
42. Gonnet GH, Baeza-Yates RA. An analysis of the Karp-Rabin string matching algorithm. Inf Process Lett 1990;34:271–4.
43. Frazee AC, Jaffe AE, Langmead B, Leek JT. Polyester: simulating RNA-seq datasets with differential transcript expression. Bioinformatics 2015;31:2778–84.
44. Cunningham F, Amode MR, Barrell D, Beal K, Billis K, Brent S, et al. Ensembl 2015. Nucleic Acids Res 2015;43:D662–9.
45. Au KF, Sebastiano V, Afshar PT, Durruthy JD, Lee L, Williams BA, et al. Characterization of the human ESC transcriptome by hybrid sequencing. Proc Natl Acad Sci Unit States Am 2013;110: E4821–30.

Supplementary Material: The online version of this article offers supplementary material (https://doi.org/10.1515/mr-2021-0016).