JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 29, Number 0, 2022
© Mary Ann Liebert, Inc.
Pp. 1–14
DOI: 10.1089/cmb.2021.0599

Waterman Special Issue

Open camera or QR reader and scan code to access this article and other resources online.

Data Set-Adaptive Minimizer Order Reduces Memory Usage in *k*-Mer Counting

DAN FLOMIN, DAVID PELLOW, and RON SHAMIR

Dedicated to Mike Waterman on his 80th birthday

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

ABSTRACT

The rapid continuous growth of deep sequencing experiments requires development and improvement of many bioinformatic applications for analysis of large sequencing data sets, including *k*-mer counting and assembly. Several applications reduce memory usage by binning sequences. Binning is done by using minimizer schemes, which rely on a specific order of the minimizers. It has been demonstrated that the choice of the order has a major impact on the performance of the applications. Here we introduce a method for tailoring the order to the data set. Our method repeatedly samples the data set and modifies the order so as to flatten the *k*-mer load distribution across minimizers. We integrated our method into Gerbil, a state-of-the-art memory-efficient *k*-mer counter, and were able to reduce its memory footprint by 30%–50% for large *k*, with only a minor increase in runtime. Our tests also showed that the orders produced by our method produced superior results when transferred across data sets from the same species, with little or no order change. This enables memory reduction with essentially no increase in runtime.

Keywords: bin mapping, *k*-mer counting, minimizer order, minimizer scheme, sequencing.

1. INTRODUCTION

HIGH-THROUGHPUT SEQUENCING (HTS) has enabled rapid progress in biological and clinical research through efficient and cheap sequencing of large genomic and transcriptomic samples. Analyzing the sequences from large HTS experiments presents computational and algorithmic challenges due to the high volume of data and the fragmented sequences generated. *k*-mer counters are a fundamental building block in some of the most basic tasks in analyzing HTS data, including genome assembly (Idury and Waterman, 1995), repeat detection, and multiple sequence alignment.

Over the past few years, several *k*-mer counters were developed. A main paradigm to speed up and reduce memory footprint of *k*-mer counting is binning (Li and Yan, 2015; Li et al., 2013; Deorowicz et al., 2015; Chikhi et al., 2016; Erbert et al., 2017). Such methods partition the entire data set into several bins, and then process each bin independently (possibly in parallel) before combining the results into the final output. Gerbil (Erbert et al., 2017) and KMC3 (Kokot et al., 2017) are popular *k*-mer counting tools that

Blavatnik School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel.

use binning, but differ in their counting approach: KMC3 sorts each bin, while Gerbil uses hash tables. BCALM2 (Chikhi et al., 2016) is a genome assembly tool that uses a *k*-mer counter similar to KMC2, an earlier version of KMC3, as a preprocess to its assembly phase.

Minimizers (Roberts et al., 2004) have been broadly used to speed up sequence analysis algorithms and to reduce disk space and memory. Given integers k and m, the *minimizer* of a k-long sequence (k-mer) is the smallest among the $k-m+1$ contiguous m-mers in it, where the smallest is determined based on a predefined order (e.g., lexicographic or random). For a longer sequence, all k-long contiguous substrings are scanned and the minimizer is selected in each one (Fig. 1).

In sequencing applications, the k-mers in the data set are partitioned using their minimizers as the key, that is, each k-mer is assigned to the partition corresponding to its minimizer. Since the same m-mer is often selected from overlapping windows, partitioning with the minimizer as key helps compress the data by representing several overlapping k-mers with the same minimizer as a longer super-k-mer in the minimizer's partition (Li et al., 2013). Often, partitions are grouped into a smaller number of bins.

Minimizer schemes have their limitations. Certain minimizers tend to appear much more than others in biological data, leading to highly unbalanced bin sizes. For example, the bin containing the minimizer $AA\ldots AA$ tends to be very large when using a lexicographic order. Several applications tried to overcome this problem by proposing alternative orders. KMC2 (Deorowicz et al., 2015) proposed the signature order, which tries to avoid certain m-mers such as $AA..A$. The Minimap2 read mapper (Li, 2018) uses a random order of the minimizers. Defining an order based on minimizers taken from a universal hitting set (UHS) was demonstrated to reduce the density and increase the mean distance between selected minimizer positions (Marçais et al., 2017). That order was shown to reduce memory in genome assembly (Ben-Ari et al., 2021).

Some methods introduced minimizer orders based on the statistics of the particular sequences of interest. Chikhi et al. (2015) showed that ordering m-mers by increasing frequency in the data dramatically reduced memory usage in assembly graph construction compared with both lexicographic and random orders. In Nyström-Persson et al. (2021), the UHS order was combined with the frequency-based order to balance bin sizes and lower memory in distributed k-mer counting. In the Winnowmap read mapper (Jain et al., 2020), the most frequent m-mers in the data set are moved to appear later in the order.

The number of distinct k-mers associated with a minimizer in a particular data set is known as the *load* of the minimizer (Ben-Ari et al., 2021). In applications that partition the data based on minimizers, hash multiple partitions into bins, and process each bin separately, it is desirable to have bins with a smaller load, to ensure low random access memory (RAM) usage, as has been argued by Li et al. (2013).

In this study, we introduce a new approach for adapting the minimizer order to the target sequence data. The method, called Adaptive Order (AdaOrder), iteratively updates the minimizer order based on an estimate of the minimizer loads in the data to reduce the maximum load. We demonstrated its ability to lower the maximum load compared with all predefined orders, except the frequency-based order.

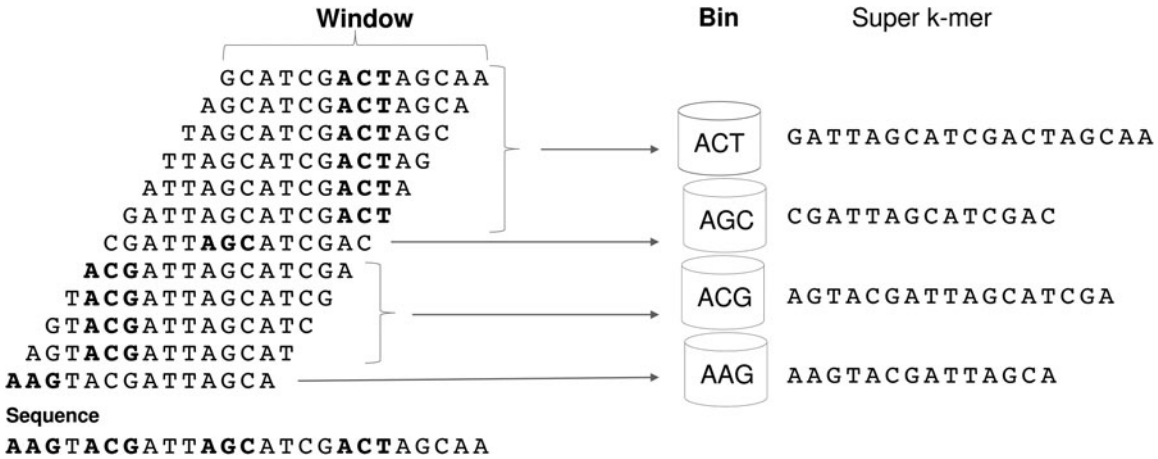

FIG. 1. Illustration of a minimizer scheme and binning. Here $k=12$ and $m=3$. The input sequence is broken into windows of length k, and in each window the m-long minimizer according to the lexicographic order (shown in bold) is selected. The k-mer is assigned to the partition whose label is the minimizer. Consecutive windows tend to select the same minimizer, and the concatenation of the consecutive windows forms the super-k-mer that is stored in the bin.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

We integrated our new order into Gerbil (Erbert et al., 2017), a leading k-mer counter, which uses signature order and was consistently the most memory efficient in a recent benchmark (Manekar and Sathe, 2018). In tests on several data sets, our implementation, called DGerbil, achieved a reduction of up to 50% in memory usage for large values of k, with only slightly higher running times compared with Gerbil. Our tests also showed that the orders produced by our method produced superior results when transferred across data sets from the same species, with little or no order change. This enables memory reduction with essentially no increase in runtime. We also implemented Gerbil with the frequency-based order (Chikhi et al., 2015), and showed that DGerbil required significantly less memory.

The code of AdaOrder and DGerbil, as well as orders produced for the studied data sets and species, is publicly available at github.com/Shamir-Lab/AdaOrder.

2. DEFINITIONS AND BACKGROUND

2.1. Basic definitions

We start with some brief basic definitions. See Deorowicz et al. (2015); Roberts et al. (2004); and Li et al. (2013) for details. A *read* is a string over the DNA alphabet $\Sigma = \{A, C, G, T\}$. An ℓ-mer is an ℓ-long string over Σ. For an ℓ-mer s with $k \leq \ell$ and $0 \leq i \leq \ell - k$, we define $s(i)$ to be the ith k-mer in s. By convention, positions in ℓ-mers start at zero.

The natural mapping $Natural : \Sigma^m \rightarrow \{0, \ldots, 4^m - 1\}$ maps m-mers to numbers according to their base 4 representation. For example: $Natural(ACG) = (012)_4 = 6$. We treat m-mers and numbers in base 4 interchangeably.

Given an m-mer x, $\bar{x}$ denotes its reverse complement. The *canonical form* of x, denoted by $Canonical(x)$, is the smaller of x and $\bar{x}$ with respect to *Natural*. For example, $Canonical(CGGT) = ACCG$. We refer to the set of all canonical m-mers as $c_m = \{Canonical(x) | x \in \Sigma^m\}$.

2.1.1. m-*mer order.* An *order* o on Σ^m is a function $o : \Sigma^m \rightarrow \mathbb{R}$. An order o can be *normalized* to $o_{norm} : \Sigma^m \rightarrow \{0, 1, \ldots, |\Sigma|^m - 1\}$ (breaking ties consistently). Here and throughout, we treat x and $\bar{x}$ as equivalent. In this notation, $o(x) \equiv o(Canonical(x))$ for any x in Σ^m and so $o(x) = o(\bar{x})$ and $o : c_m \rightarrow \mathbb{R}$.

2.1.2. Minimizers. A *minimizer* of a sequence s with respect to order o is the smallest m-long contiguous substring z in s according to o. We also call z the o-minimizer m-mer in s.

Within a sequence, m-mer x is smaller than m-mer y according to an order o if: (i) $o(x) < o(y)$; or (ii) $o(x) = o(y)$ and $Natural(x) < Natural(y)$; or (iii) $o(x) = o(y)$ and $Natural(x) = Natural(y)$ and x occurs to the left of y in s. In other words, we break ties according to *Natural*, and if needed also by choosing the leftmost appearance of the least m-mer.

A *minimizer scheme* is a function $f_{o,m,k} : \Sigma^k \rightarrow [0 : k - m]$ that selects the start position of the o-minimizer m-mer in every sequence of length k (Fig. 1).

2.1.3. Lexicographic order. We define the lexicographic order to be as follows: $o_{lexico}(x) = Natural(x)$.

2.1.4. Signature order. The following *signature order* was proposed in Deorowicz et al. (2015) as an alternative to the lexicographic order. Each m-mer that either starts with AAA or ACA or contains AA as a substring is called *bad*. The rest of the m-mers are called *good*. The good m-mers are ordered lexicographically, and all of them are defined to be smaller than the bad m-mers. The bad m-mers are not ordered and are all mapped to the same value.

We define a variant of the signature order as follows:

$$o_{sig}(x) = \begin{cases} o_{lexico}(x) & x \text{ is good} \\ o_{lexico}(x) + 4^m & x \text{ is bad} \end{cases}$$

Note that unlike in Deorowicz et al. (2015), we do order the bad m-mers.

2.1.5. Random order. We define a pseudorandom order by performing xor operation between the binary representations of a random integer mask α and of the lexicographic order of m-mers. We denote $\oplus$ as the bitwise xor operation. The random order is formally defined as follows:

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

$$o_{random}(s, \alpha) \equiv o_\alpha(s) = o_{lexico}(s) \oplus \alpha$$

The xor operation is a bijection, and therefore, if $s \neq s'$ then $o_\alpha(s) \neq o_\alpha(s')$.

2.1.6. Frequency order. The *frequency order* (Chikhi et al., 2015) is a minimizer order based on statistics collected from a specific data set. It orders the m-mers in increasing frequency order according to the data set. For our analysis, we defined it as follows: Given a data set D, we count m-mer appearances in it, and set

$$o_{frequency}(D, x) = count(D, x)$$

where *count* is the number of times x appeared in D.

*2.1.7. Partitions, partition size, bins, and super-*k*-mers.* In *minimizer partitioning*, each k-mer in the input data set is assigned to a partition (e.g., a file on disk) according to its minimizer. Each partition corresponds to only one minimizer and vice versa. When the number of partitions is too large, partitions are often combined into a smaller number B of groups called *bins*, where each bin contains the k-mers of several minimizers, and each minimizer corresponds to only one bin. The *size* of a partition or a bin is the total number of characters in all k-mers in it.

Several consecutive k-mers that share the same minimizer can be compressed together. A *super-k-mer* (Li et al., 2013) is the longest substring for which all k-mers have the same minimizer. Figure 1 shows an example of partitions and super-k-mers.

2.2. Quality criteria for minimizer schemes

Following Ben-Ari et al. (2021) and Nyström-Persson et al. (2021), we define quality criteria for a minimizer order on a data set. They are meant to indicate how evenly the k-mers of a data set are distributed across the minimizers.

2.2.1. Load. Given a minimizer scheme $f_{o,m,k}$ and a data set D, the *load* of $x \in c_m$ is the number of distinct k-mers in D for which x is the minimizer (Ben-Ari et al., 2021). We denote it by $l(x, D)$. Similarly, when binning is used, we define the bin's load to be the number of distinct k-mers in it.

We also define the *relative load* of $x \in c_m$ over data set D to be x's load divided by the total number of distinct k-mers in the data set. We denote it by $r(x, D)$.

The *maximum* (relative) load is the (relative) load of the minimizer with the highest (relative) load. Maximum (relative) load is defined analogously for bins.

2.2.2. Unevenness. Unevenness aims to describe how balanced a minimizer scheme is over a particular data set.

Given a minimizer scheme $f_{o,m,k}$ and a data set D, we define its unevenness to be:

$$U(f_{o,m,k}, D) = \frac{1}{|c_m|} \cdot \sum_{i \in c_m} \left(r(i, D) - \frac{1}{|c_m|} \right)^2.$$

Recall that $|c_m|$ is the number of possible m-long minimizers, and thus, the unevenness is the mean squared difference from the uniform distribution of k-mers across minimizers.

2.3. The Gerbil k-mer counter

Gerbil (Erbert et al., 2017) is a memory-efficient k-mer counter. It bins k-mers according to their minimizers, using the signature order in its minimizer scheme, and merges adjacent k-mers sharing the same minimizer into super-k-mers. Then the k-mers in each bin are counted independently using a hash table.

2.4. Mapping minimizers to bins in Gerbil and KMC3

k-mer counters use a small fixed number of bins (e.g., 512 in Gerbil) and assign multiple partitions to the same bin. The mapping of minimizers into bins could be performed randomly (e.g., using a random hash), but many k-mer counters use heuristics that aim to better balance the loads of the bins. We note that

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

achieving an optimal distribution of minimizers to bins is equivalent to the NP-Hard problem of optimal identical machines scheduling with minimum makespan (Garey and Johnson, 1979). There exist complex heuristics and polynomial time approximation schemes for this problem (Vazirani, 2003), however, in practice k-mer counting tools use simple heuristics without guarantees, in the interest of runtime and simplicity.

Gerbil tries to group together minimizers that appear earlier in the order (and thus are intuitively likely to have higher load) with those that appear later. Specifically, each minimizer is added to a new bin in increasing signature order. The order of the bins to which minimizers are added is reversed each time a minimizer has been added to every bin.

KMC3 (Kokot et al., 2017) samples a portion of the data set to estimate the partition size of each minimizer. After collecting the statistics, it adds 1000 to each partition size and then sorts the minimizers by their sampled counts from the largest to the smallest. Minimizers are mapped to the current bin until its size exceeds the average remaining bin size, and then, a new bin is opened, and the process continues in that bin. An outline of the process is described in Supplementary Algorithm S1 (Supplementary Data).

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

3. METHODS

We developed an algorithm called AdaOrder that constructs a minimizer order with low maximum load for a given data set, based on statistics collected on the data set. We also implemented a sampling process that estimates the total size of each partition. These estimates are used to determine an efficient mapping of minimizers to bins, similar to KMC3. The process is schematically as follows. (i) Compute a minimizer order using AdaOrder; (ii) estimate the minimizer partition sizes by sampling; and (iii) map minimizers to bins using the estimates from (ii). We integrated this process into Gerbil and call the modified algorithm DGerbil.

3.1. AdaOrder

AdaOrder is a heuristic that aims to produce a minimizer order with low maximum minimizer load in a given data set. AdaOrder starts with the scheme $f_{o_{init},m,k}$ and works in R rounds. In each round, it samples N k-mers from the data set to capture a minimizer with a large load. A single round proceeds as follows: (i) iterate over the data set's reads sequentially; (ii) for each read, scan through all of its k-mers and identify minimizers; (iii) compute the load of each minimizer according to the sampled k-mers; and (iv) after sampling at least N k-mers, identify the minimizer z with the highest sample load. In case of ties, choose the lexicographically smallest minimizer. Alter the current order by increasing the order of z by $p \cdot 4^m$, where p is a penalty factor. This makes z less likely to be chosen as a minimizer, thus lowering its load. Algorithm 1 describes AdaOrder.

Algorithm 1. AdaOrder

Input: A set of reads $D=(S_0, S_1, \ldots, S_{M-1})$, where $|S_i| \geq k$.
A minimizer scheme $f_{o_{init},m,k}$. R - number of rounds, N - number of samples per round, p - penalty factor.
Output: A minimizer scheme $f_{o',m,k}$.

$read \leftarrow 0$
$o \leftarrow o_{init}$
for *round* from 1 to R **do**
  $sampled \leftarrow 0$
  for all m-mers y **do** $H_y \leftarrow$ an empty hash table
  while $sampled < N$ **do**
    for i from 0 to $|S_{read}| - k$ **do**
      $x \leftarrow$ the o-minimum m-mer of $S_{read}(i)$
      Insert $Canonical(S_{read}(i))$ into H_x
      $sampled \leftarrow sampled + 1$
    $read \leftarrow read + 1$
  $z \leftarrow$ m-mer with largest $|H_z|$
  $o(z) \leftarrow o(z) + p \cdot 4^m$
Return $f_{o,m,k}$

3.2. Mapping minimizers to bins

We first estimate each partition size through sampling (Supplementary Algorithm S2), and then map minimizers to bins using these estimates. The bin mapping algorithm is shown in Algorithm 2; it heavily relies on KMC3's implementation.

3.3. DGerbil

DGerbil is based on the memory-efficient k-mer counter Gerbil. It integrates AdaOrder initialized with signature order and our bin mapping algorithm into Gerbil. The use of AdaOrder aims to lower the maximum bin load, to lower the RAM usage. The algorithm for DGerbil is shown in Algorithm 3. Gerbil's binning method and counting method are described in Section 2.3.

Algorithm 2. BinMapping

Input: Data set D. M - minimizer length. B - number of bins.
Output: A mapping of the partitions to bins.
1: Sample D and obtain a count $c(m)$ for each minimizer m
2: Map minimizers with $c(m)=0$ evenly across B bins
3: Sort minimizers with $c(m) \neq 0$ in decreasing order of $c(m)$
4: $nb=B$; $Tot=\sum_m c(m)$ // #remaining bins; total size of remaining partitions
5: $m=1$ // the minimizer with largest size that is still unmapped
6: **for** *bin* from 1 to B **do**
7: $Mean=Tot/nb$ // mean size of the remaining bins
8: $size=0$ // filled size of the current bin
9: **while** $size < Mean$ **do**
10: add $c(m)$ to size, map m to bin *bin* and increase m by 1
11: $Tot=Tot-size$; $nb=nb-1$
12: Return the mapping of partitions to bins and the filled bin sizes

3.4. FGerbil

We implemented a variant of Gerbil that uses frequency order, called FGerbil. It first counts how many times each m-mer appeared in the data set, and orders m-mers by their frequency. It then uses the resulting frequency order within Gerbil, and performs bin mapping as DGerbil does.

3.5. Source code

The methods and code base to run AdaOrder and DGerbil are available at github.com/Shamir-Lab/AdaOrder. AdaOrder is coded in Java, while DGerbil is a modification of Gerbil's C++ code.

In our implementation of AdaOrder, some optimizations were introduced to Algorithm 1 for a faster running time. See Supplementary Section S4 for details.

Algorithm 3. DGerbil

Input: A set of reads $D=(S_0, S_1, \ldots, S_{M-1})$, where $|S_i| \geq k$. m - minimizer length, R - number of rounds, N - number of samples per round, p - penalty factor, B - number of bins, E - number of k-mers to sample.
Output: k-mer counts
1: Generate minimizer ordering, *order*, using AdaOrder initialized with o_{sig} (**Algorithm 1**)
2: Collect minimizer partition sizes, *stats*, using MinimizerStats (**Supplementary Algorithm S2**) and *order*
3: Create minimizer to bins mapping, *mapping*, using BinMapping (**Algorithm 2**) and *stats*
4: Apply Gerbil's binning method over D using *order* and *mapping*
5: Use Gerbil's counting method to count k-mers in each bin
6: Return k-mer counts

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

TABLE 1. CHARACTERISTICS OF THE BENCHMARK DATA SETS

Data set	*Size (10^9 bytes)*	*Average read length*	*Species*
HS1	292	151	*Homo sapiens*
HS2	347	202	*H. sapiens*
AT	72.7	4804	*Arabidopsis thaliana*
FW	406	300	Freshwater metagenome
FV	14.1	508	*Fragaria vesca*
NC	45.9	7778	*Neurospora crassa*
DM	10.7	152	*Drosophila melanogaster*
MB	198	101	*Musa Balbisiana*

Note that the sequence read archive (SRA) toolkit merges pair-end reads into a single read by default when downloading, and therefore, the pair-end reads of HS2, MB, DM, FV, and FW were merged and the reported read length is after the merge.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

4. RESULTS

We tested the performance of AdaOrder and several popular orders on multiple data sets, and also compared the performance of DGerbil (which uses AdaOrder), FGerbil (which uses frequency), and the original Gerbil.

We used eight data sets; seven were those used in a recent *k*-mer counter benchmark (Manekar and Sathe, 2018) and the eighth was a large freshwater metagenomic data set [FW (Mehrshad et al., 2018), SRR6787039]. See Table 1 for the properties of the data sets. Both AT and NC are long read data sets collected using the PacBio technology.

Throughout this section, the default parameters used for AdaOrder unless stated otherwise were $o_{init} = o_{sig}$, $R = 10^4$, $N = 10^5$, and $p = 0.01$. The choice of these parameters is discussed in Supplementary Section S3. For the frequency order, we collected statistics over the entire data set. Minimizer length was always $m = 7$.

4.1. AdaOrder reduces maximum load and unevenness

We applied the lexicographic, signature, random, frequency, and AdaOrder orders on four large data sets for $k = 28$ and 55. We used the same values of k as in Manekar and Sathe (2018). In the random order, we used a different random mask for each generated random order. Maximum load and unevenness results are shown in Tables 2 and 3, respectively. Frequency order consistently had the lowest maximum load and unevenness followed by AdaOrder, while the predefined orders were substantially worse.

We also calculated the distribution of load across minimizers for each order. In Figure 2a, we plot the load of the 1000 minimizers with the highest load on HS2 using $k = 55$ for the different orders. AdaOrder and frequency distributed the *k*-mers much more evenly across these minimizers than the other orders, in line with the results in Tables 2 and 3. Figure 2b shows the cumulative distribution of the load for all minimizers. AdaOrder and frequency order did a good job in balancing the top loads compared with the other orders. For example, when using signature, $\sim 20\%$ of *k*-mers were covered by the top 20 minimizers, while AdaOrder used the 100 top minimizers to cover $\sim 20\%$ of the *k*-mers.

TABLE 2. MAXIMUM LOAD $\times 10^{-6}$

Data set	*k*	*Lexicographic*	*Random*	*Signature*	*AdaOrder*	*Frequency*
HS1	28	78.5	20.1	14.6	5.7	**3.0**
	55	157.1	51.7	36.7	10.2	**4.8**
HS2	28	83.7	22.1	15.6	5.8	**3.1**
	55	175.3	60.0	41.7	9.9	**5.1**
FW	28	58.1	54.0	27.1	11.0	**5.3**
	55	112.3	70.2	58.8	19.0	**8.6**
AT	28	35.1	4.7	12.0	0.6	**0.5**
	55	6.0	1.2	3.7	0.8	**0.3**

Results with the lowest maximum load are shown in bold.

TABLE 3. UNEVENNESS $\times 10^7$

Data set	*k*	*Lexicographic*	*Random*	*Signature*	*AdaOrder*	*Frequency*
HS1	28	4.6	2.6	2.5	1.0	**0.4**
	55	11.4	4.5	5.2	1.6	**0.6**
HS2	28	4.6	2.0	2.5	1.0	**0.4**
	55	11.6	3.7	5.2	1.5	**0.5**
FW	28	1.8	2.1	1.8	0.9	**0.3**
	55	4.2	4.4	3.8	1.9	**0.6**
AT	28	18.0	3.1	4.0	0.7	**0.2**
	55	168.5	15.9	63.4	5.7	**2.2**

Results with the lowest unevenness are shown in bold.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

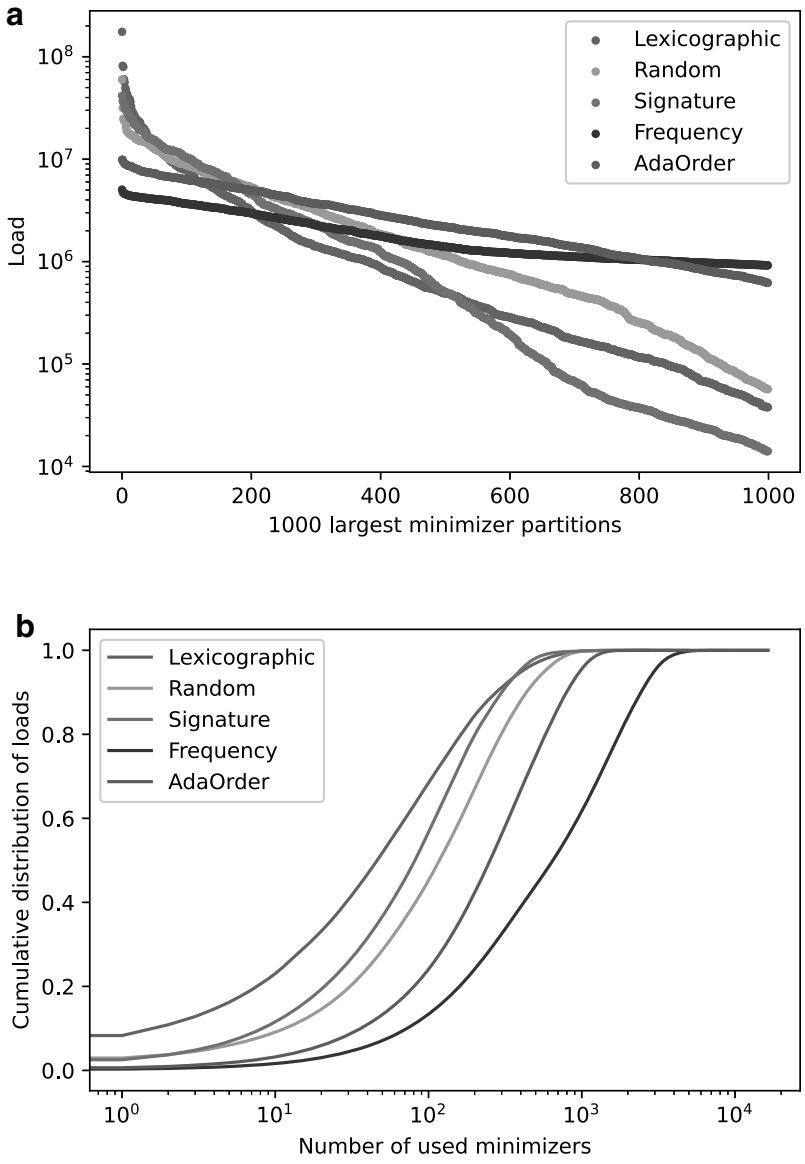

FIG. 2. Distribution of loads across minimizers. Results are for HS2 with $k=55$. **(a)** Load of the 1000 minimizers with highest load. **(b)** Cumulative distribution of the load of all minimizers. In both figures, minimizers were sorted in decreasing load for each order.

The *number* of minimizers used also differed substantially between orders: 7731 for frequency, 4940 for AdaOrder, 4939 for signature, 2694 for lexicographic, and 3828 for the random order. Interestingly, AdaOrder achieved much lower and more even maximum loads than the signature order while still using a very similar number of minimizer partitions, whereas frequency order used many more minimizers to achieve better performance.

4.2. *The effect of* R, N, *and* p *on maximum load and unevenness*

We explored the impact of each of the input parameters R, N, and p of AdaOrder on the maximum load and unevenness. This analysis can help in the choice of parameters in AdaOrder. In Supplementary Section S1, we performed a statistical analysis of the accuracy of sampling-based load estimation, which sheds light on the recommended size of N. We tested various combinations of R, N, and p. In each test, we fixed two

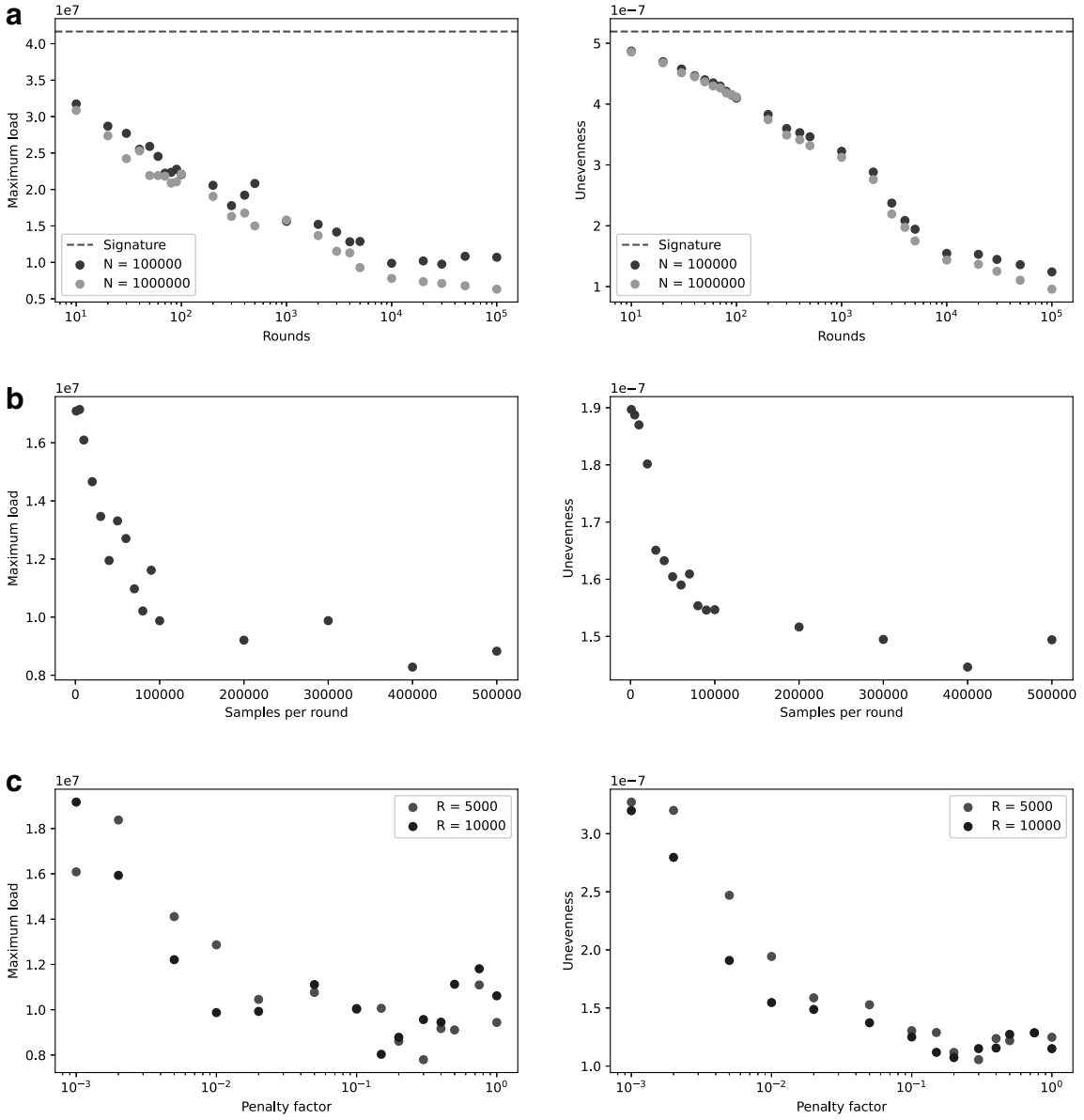

FIG. 3. Maximum load and unevenness when varying parameters R, N, and p in AdaOrder. Each graph shows the performance when varying a single parameter. **(a)** Impact of varying R, the number of rounds. The horizontal dashed line corresponds to the maximum load and unevenness value of the signature order with which it is initialized. **(b)** Impact of changing the sample size N. **(c)** Impact of the penalty factor p.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

parameters to their default values and varied the third. All the tests were performed on the HS2 data set with $k=55$. We chose HS2 for this experiment since it is a large data set and since we were particularly interested in the performance for *Homo sapiens*.

Figure 3a shows the effect of varying R. We observe that both maximum load and unevenness drop drastically from the baseline within relatively few rounds (note that the x-axis is logarithmic). The load plateaus after around 10^4 rounds, while unevenness keeps decreasing slowly.

Figure 3b shows the effect of varying N. Both maximum load and unevenness decrease as N gets larger. This demonstrates the benefit of choosing a large sample.

Figure 3c shows the effect of varying p. The load decreases first as p increases, but no clear trend is observed for $p \geq 0.1$, with substantial jumps in the load. A similar trend but with lower variability is observed for unevenness. The trends are similar when using 5000 and 10,000 rounds. The high variability for high values of p is likely due to large changes in the order in the final rounds after all of the top loads have already been made relatively even. Supplementary Figure S1 shows a similar analysis for $N = 10^4 - 10^8$ and different rounds.

Based on the analysis above and the theoretical analysis in Supplementary Section S1, we established recommended parameters for running AdaOrder. See Supplementary Section S3 for details.

4.3. Transferring the order across data sets

We wished to see whether an order produced on one data set can assist in creating orders for the other data sets from the same species, either by running it as is, or by using it as a starting order in the optimization for the other data sets. We reasoned that if this is the case, then it could speed up the process of producing good orders for other data sets from the same species.

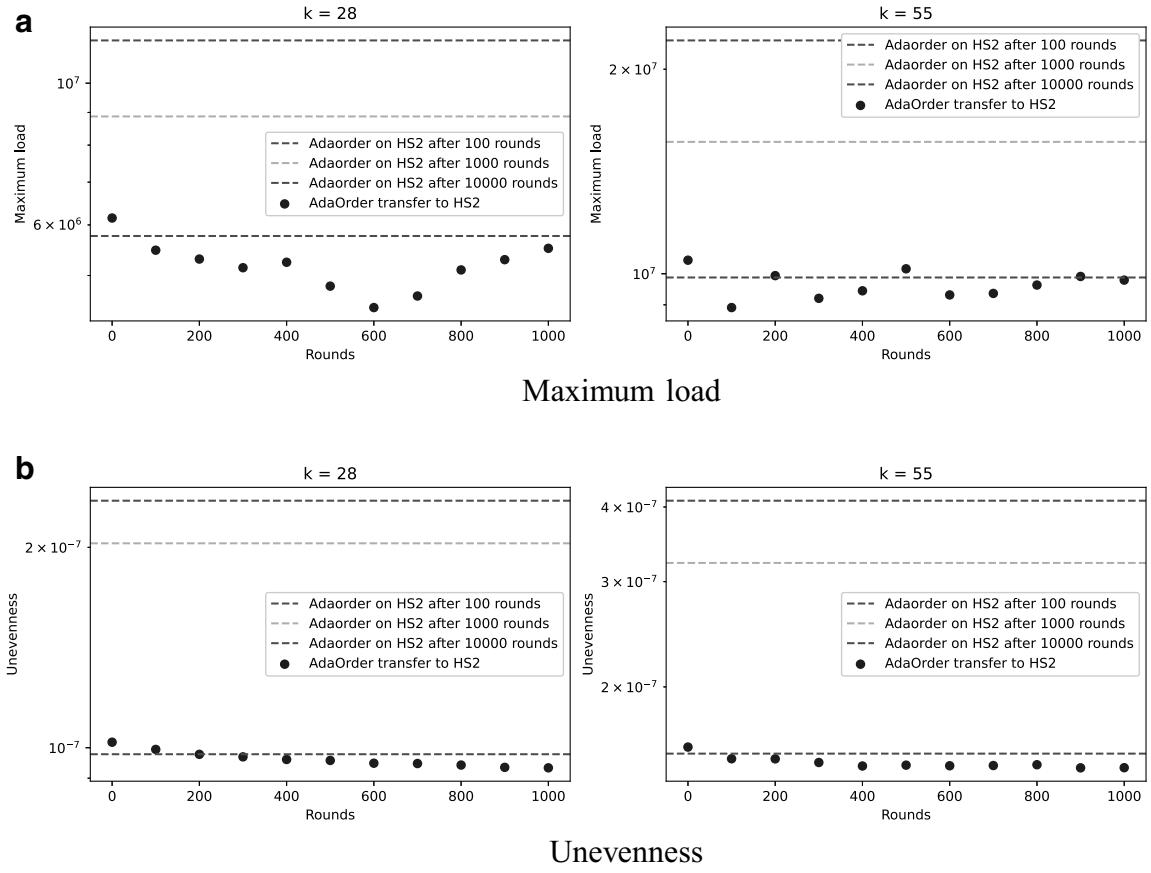

FIG. 4. Order transfer from HS1 to HS2. **(a)** Maximum load; **(b)** unevenness. The value at round 0 corresponds to applying the HS1 order as is on HS2. The dashed lines correspond to the values obtained by running AdaOrder on HS2 with the default initialization, for different number of rounds.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

Suppose we have a source data set *src* and a destination data set *dst*. We apply AdaOrder on *src* and use that order on *dst*. We performed two experiments. One where *src* was HS1 and *dst* was HS2, and another where *src* was HS2 and *dst* was HS1. We ran AdaOrder with $k = 28$ and $k = 55$ over *src* and tested the efficiency of the minimizer order it produced over *dst*. We also ran AdaOrder over *dst*, initialized with the order produced by AdaOrder for *src*.

The results are shown in Figures 4 and 5. We see that transferring the order works similarly to the AdaOrder results produced for the destination data set independently of *src* (the dashed green line). Moreover, starting from the *src* order, the algorithm can improve unevenness and maximum load within far fewer optimization rounds compared with running AdaOrder for *dst* from scratch (<1,000 vs. 10^4). We conclude that the order can be transferred across data sets from the same species, saving most of the running time and preserving the order quality.

4.4. DGerbil reduces memory usage

We compared the performance of Gerbil, DGerbil, and FGerbil. All algorithms were run with 512 bins and $k = 28, 55, 70, 90$ on each data set [28 and 55 were used in Manekar and Sathe (2018)]. All the experiments were measured on a 128-core server with 64 3.35 GHz CPUs and 1000GB of RAM (AMD EPYC 7702). We ran all algorithms with 12 threads, the same number of threads as in Manekar and Sathe (2018). The storage used in our experiments is an external NFS storage, and all input/output (IO) operations were performed against it (e.g., reading data sets and writing temporary and output files). The RAM usage of AdaOrder was externally limited to a maximum of 1.6GB (we used Java's Xmx flag to limit the program's heap size). The goal of the limit was to keep memory small enough while allowing AdaOrder to run fast.

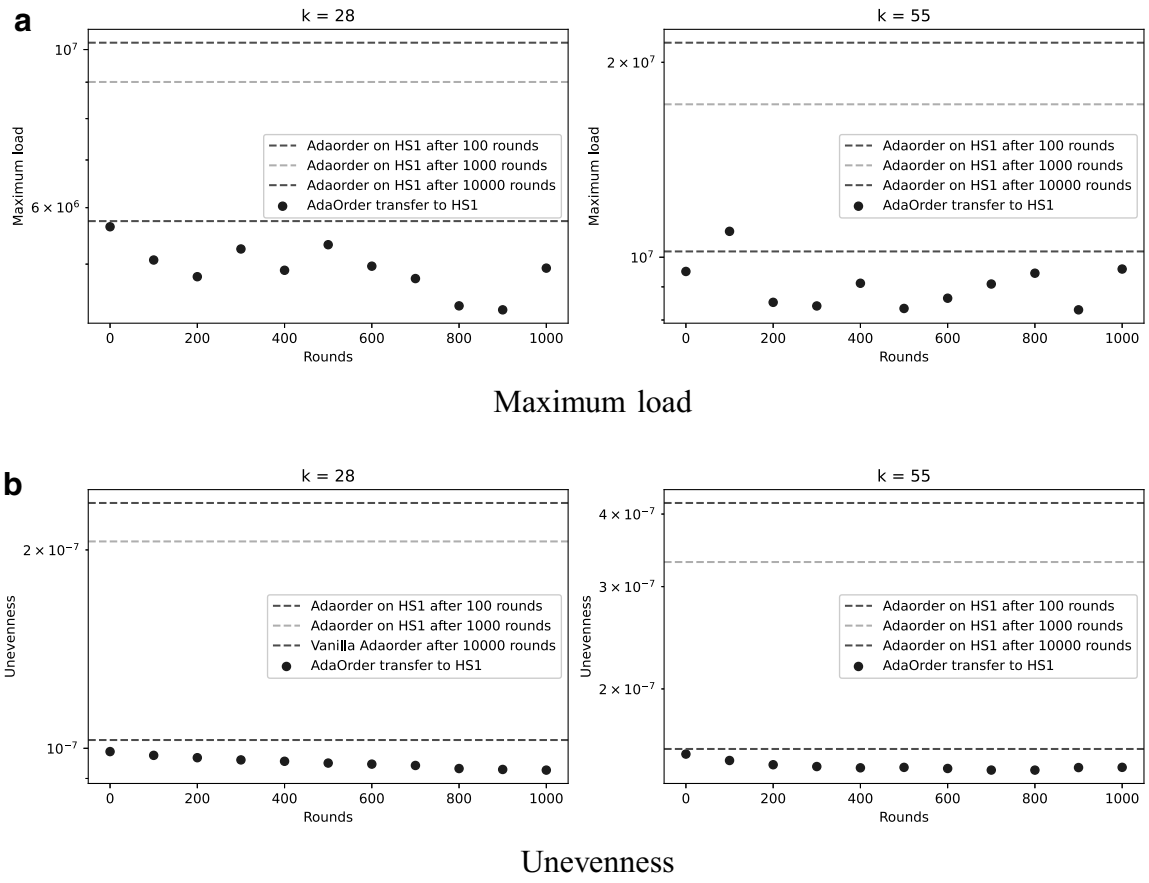

FIG. 5. Order transfer from HS2 to HS1. **(a)** Maximum load; **(b)** unevenness. The value at round 0 corresponds to applying the HS2 order as is. The dashed lines correspond to the values obtained by running AdaOrder on HS1 with the default initialization, for different number of rounds.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

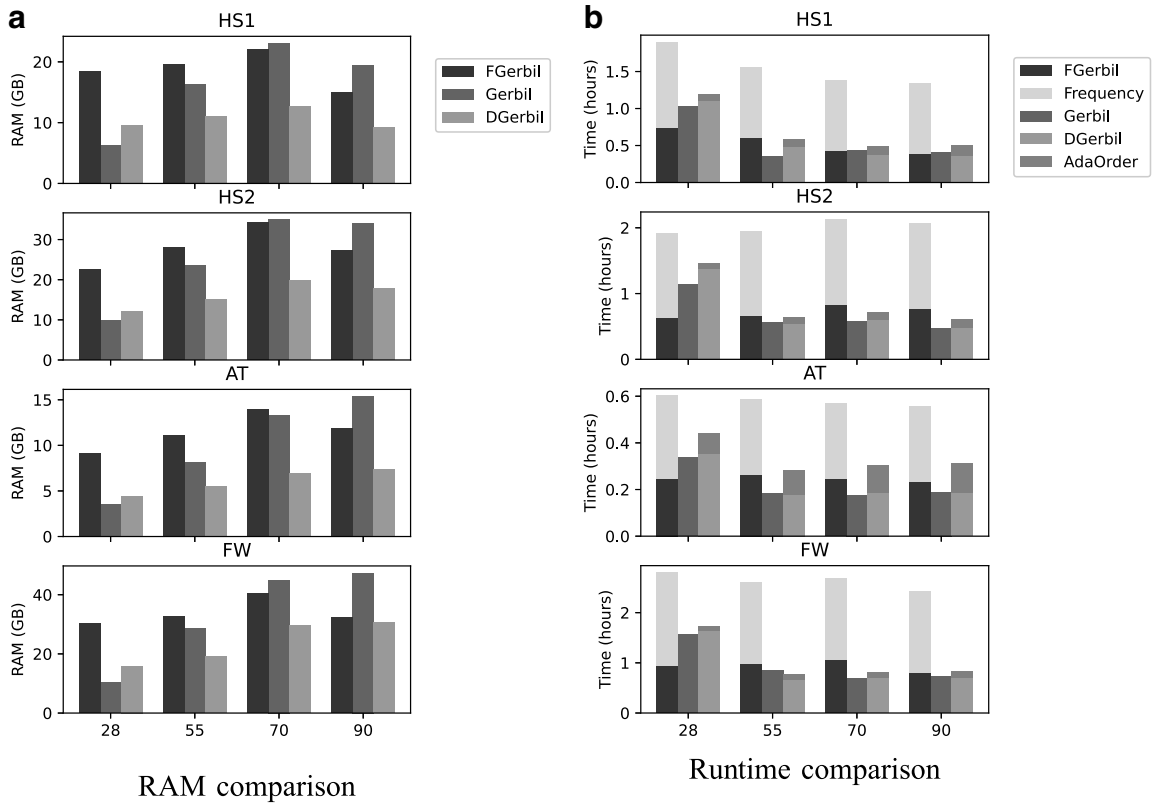

FIG. 6. Performance of Gerbil, FGerbil, and DGerbil. **(a)** RAM comparison; **(b)** runtime comparison. In **(b)**, for DGerbil and FGerbil, AdaOrder and frequency times for creating the order and collecting binning statistics are shown separately. AT; DM; FV; FW; MB; NC; RAM.

The results for the four data sets with the highest RAM usage are shown in Figure 6. The results for the other four data sets are in Supplementary Figure S2. The figures show the runtime and RAM usage on each data set. For DGerbil (respectively, FGerbil) the time of AdaOrder (respectively, frequency order) is shown separately. DGerbil used consistently lower RAM except when using the smallest value of k (28). All tools tended to use more memory as k increased. Similar RAM trends were observed for the other data sets (Supplementary Fig. S2), although absolute numbers were lower.

In terms of time, it took an average of 6 m 50 seconds to run AdaOrder and to collect statistics for binning. This runtime was roughly constant as the sampling process does not depend on the data set size. For frequency it took on average about 39 minutes to finish collecting statistics for the order and for the binning. For the smaller data sets shown in Supplementary Figure S2, the runtime of Gerbil was already very low, and therefore, the time to run AdaOrder or frequency dominated that of the k-mer counting.

5. DISCUSSION

We developed AdaOrder, an algorithm for creating an improved minimizer order on a given data set by repeatedly adjusting the order to reduce the maximum load. AdaOrder significantly reduced maximum load and unevenness compared with the signature order and with the other predefined orders. The performance of the frequency order was even better, but it did not translate to a better k-mer counting algorithm (see below). AdaOrder was integrated into the k-mer counter Gerbil together with a k-mer sampling method to map minimizers into bins. This reduced memory usage by 30%–50% of Gerbil for large k.

AdaOrder has a roughly constant runtime of less than 7 minutes regardless of the size of the data set. For small data sets, the absolute reduction in the memory usage is minor, and the running time of AdaOrder dominates the k-mer counting process, so using AdaOrder is not advantageous. The main advantage of AdaOrder is on larger data sets, especially when longer k-mers are used. In these cases, the memory savings are substantial, while the additional time required to run AdaOrder is relatively minor.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

DGerbil outperformed Gerbil in memory, but required a bit more time. By transferring an order precomputed by AdaOrder on another data set of the same species, most or all of this time can be saved, thus matching Gerbil's time and cutting memory by 30% – 50% on large k values.

Interestingly, applying only one of our modifications to Gerbil individually—using AdaOrder instead of signature order, or mapping minimizers to bins based on k-mer sampling statistics—actually *increased* the memory usage (results not shown). Hence, we suspect that multiple factors (maximum load, the algorithm for bin mapping, and details of the implementation) together determine the actual memory usage of Gerbil.

Specific implementation details of Gerbil may also explain some of the other results we observed. For example, Gerbil decides in advance what hash table size to use for a bin based on the number of k-mers written to it, and on the ratio of distinct k-mers to total k-mers in the previous bin. Choosing a hash table size that is too big wastes memory, while choosing one that is too small increases running time. Since this heuristic and its parameters were optimized in Gerbil, this may explain why Gerbil has a better performance for the smallest k values tested ($k = 28$). Similar considerations may explain why the frequency order fails to improve Gerbil. The distribution of minimizer loads is very different from what Gerbil expects and is implemented to optimize.

Several areas for improvement and open questions remain. Most importantly, can one improve the algorithm by dynamically choosing values for the parameters R, N, and p?

(i) Our analysis suggests that the number of samples per round should not be fixed throughout the entire process, as it depends both on the maximum relative load and on the difference between the top relative loads, both of which decrease as the algorithm progresses. (ii) Having a stopping condition for the entire process instead of a fixed number of rounds could be beneficial, as we observed that AdaOrder continues to alter the order even after it is no longer beneficial. (iii) One may want to penalize the maximum load minimizer in a round as a function of how large the estimated maximum relative load is (i.e., if the maximum load is larger, then we would like p to be larger). Together, some or all of these ideas could improve the algorithm further.

The frequency order achieved superior maximum load and unevenness results compared with AdaOrder. However, its use in Gerbil required more RAM compared with DGerbil. An interesting topic for future research is the design of a k-mer counter with low memory usage based on the frequency order. Leveraging frequency's potential in this task may lead to an even more memory-efficient and faster k-mer counter.

We have demonstrated the utility of directly optimizing the maximum load of a minimizer order in binning applications. Using AdaOrder in Gerbil further improved its RAM usage, especially for longer k-mers. This approach has the potential to reduce the memory footprint of other sequence analysis algorithms on large data sets.

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.

ACKNOWLEDGMENTS

We thank Nimrod Rappoport and Jonathan Schifter for helpful advice.

AUTHOR DISCLOSURE STATEMENT

The authors declare they have no conflicting financial interests.

FUNDING INFORMATION

The study was supported, in part, by the Israel Science Foundation (Grant No. 3165/19, within the Israel Precision Medicine Partnership program, and Grant No. 1339/18), by Len Blavatnik and the Blavatnik Family foundation, and by the Zimin Institute for Engineering Solutions Advancing Better Lives. D.P. was supported, in part, by a fellowship from the Edmond J. Safra Center for Bioinformatics at Tel-Aviv University and, in part, by a PhD fellowship from the Israel Ministry of Immigrant Absorption.

SUPPLEMENTARY MATERIAL

Supplementary Data

REFERENCES

Ben-Ari, Y., Flomin, D., Pu, L., et al. 2021. Improving the efficiency of de Bruijn graph construction using compact universal hitting sets. Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB'21. Association for Computing Machinery, New York.

Chikhi, R., Limasset, A., Jackman, S., et al. 2015. On the representation of de Bruijn graphs. *J. Comput. Biol*. 22, 336–352.

Chikhi, R., Limasset, A., and Medvedev, P. 2016. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*. 32, i201–i208.

Deorowicz, S., Kokot, M., Grabowski, S., et al. 2015. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*. 31, 1569–1576.

Erbert, M., Rechner, S., and Müller-Hannemann, M. 2017. Gerbil: A fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol. Biol*. 12, 9.

Garey, M.R., and Johnson, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, 1st ed. W.H. Freeman, New York.

Idury, R.M., and Waterman, M.S. 1995. A new algorithm for DNA sequence assembly. *J. Comput. Biol*. 2, 291–306.

Jain, C., Rhie, A., Zhang, H., et al. 2020. Weighted minimizer sampling improves long read mapping. *Bioinformatics*. 36(Suppl. 1), i111–i118.

Kokot, M., Długosz, M., and Deorowicz, S. 2017. KMC 3: Counting and manipulating k-mer statistics. *Bioinformatics*. 33, 2759–2761.

Li, H. 2018. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics*. 34, 3094–3100.

Li, Y., Kamousi, P., Han, F., et al. 2013. Memory efficient minimum substring partitioning. *Proc. VLDB Endowment*. 6, 169–180.

Li, Y., and Yan, X. 2015. MSPKmerCounter: A fast and memory efficient approach for k-mer counting. *arXiv preprint arXiv:1505.06550*.

Manekar, S.C., and Sathe, S.R. 2018. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*. 7, 10.

Marçais, G., Pellow, D., Bork, D., et al. 2017. Improving the performance of minimizers and winnowing schemes. *Bioinformatics*. 33. i110–i117.

Mehrshad, M., Salcher, M.M., Okazaki, Y., et al. 2018. Hidden in plain sight—highly abundant and diverse planktonic freshwater chloroflexi. *Microbiome*. 6, 1–13.

Nyström-Persson, J., Keeble-Gagnère, G., and Zawad, N. 2021. Compact and evenly distributed k-mer binning for genomic sequences. *Bioinformatics*. 37, 2563–2569.

Roberts, M., Hayes, W., Hunt, B.R., et al. 2004. Reducing storage requirements for biological sequence comparison. *Bioinformatics*. 20, 3363–3369.

Vazirani, V.V. 2003. Minimum makespan scheduling, 79–83. *In* Vazirani, V.V., ed. *Approximation Algorithms*. Springer, Berlin, Heidelberg.

Address correspondence to:
Prof. Ron Shamir
Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv 69978
Israel

E-mail: rshamir@tau.ac.il

Downloaded by Tel Aviv University from www.liebertpub.com at 06/06/22. For personal use only.