Tel-Aviv University

Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Dataset-adaptive minimizer order reduces memory usage in k-mer counting

Thesis submitted in partial fulfillment of graduate requirements for
The degree "Master of Sciences" in Tel-Aviv University
School of Computer Science

By

Dan Flomin

Prepared under the supervision of

Prof. Ron Shamir

March 2022

Acknowledgments

I want to thank to all the people that helped me reach this moment in my academic path, and helped me achieve a meaningful experience in researching in the field of Bioinformatics.

I want to thank Prof. Ron Shamir, who welcomed me into his research group and supervised over our research. Ron helped me greatly in exploring different research fields, and in finding the topic which I found very fascinating to research. The path we explored together ignited an eager wish to deepen my understanding in my research and to make it meaningful to other researchers in the field as well. One time after the other Ron pushed me to broaden our understanding, and to write a research paper in a very high standard. The last two years with Ron developed myself as a researcher and as a person, and I thank him dearly for this opportunity.

I want to thank David Pellow, my fellow lab mate and researcher. David mentored me from the very first moments as a member of the lab. At first, he helped me understand complex academic materials, and later on when I already started to research myself, he helped and advised me in my research. At some point during my research, I knew that joining forces with David in my research could be very meaningful to the research, and a lot more fun. David is a humble, kind, smart and amazing person, and I had the best time researching with him, learning from him. I thank him a lot and appreciate the opportunity to research with him.

Early in my time as a lab member, I joined Yael Ben-Ari in her research. I learned a great deal besides Yael as a very young researcher. I want to thank Yael for her part in me becoming a researcher, her welcoming me warmly to her research, and for our friendship.

I want to thank Nimrod Rappoport for always being there for me when I needed is help and advice. Nimrod had a great impact on me as a TA and has a great part in me wanting to research in Bioinformatics and pursuing to join Ron's lab.

I want to thank my lab members: Lianrong, Tom, Dan, Hagai, Hadar, Naama, Omer, Yonatan, Eran, Ron, Roi and Maya. All of them took a great part in the last two years, academically and as friends.

Gilit Zoher-Oren supported me at all times, with any matter, and I thank her very much for doing so, and doing so very kindly.

Last but not least, I want to thank for my parents Orit and Dror. They raised me to be the person I am today, who was able to take part in academic research and to make the best out of it. They supported me in every path I chose for myself, and drive me towards excellence in my own way and terms. Thank you.

Abstract

The rapid, continuous growth of deep sequencing experiments requires development and improvement of many bioinformatics applications for analysis of large sequencing datasets, including k-mer counting and assembly. Several applications reduce RAM usage by binning sequences. Binning is done by employing 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 dataset. Our method repeatedly samples the dataset 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 minor increase in runtime. Our tests also showed that the orders produced by our method produced superior results when transferred across datasets from the same species, with little or no order change. This enables memory reduction with essentially no increase in runtime.

Contents

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 (Benson and Waterman, 1994), and multiple sequence alignment (Carrillo and Lipman, 1988).

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 (Chikhi et al., 2016; Deorowicz et al., 2015; Erbert et al., 2017; Li et al., 2015, 2013). Such methods partition the entire dataset 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 use binning, but differ in their counting approach: KMC3 sorts each bin, while Gerbil employs hash tables. BCALM2 (Chikhi et al., 2016) is a genome assembly tool that uses a k-mer counter similar to KMC2 (Deorowicz et al., 2015), an earlier version of KMC3, as a pre-process 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.

In sequencing applications, the k-mers in the dataset are partitioned using their minimizers as the key, i.e., 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 subsequently 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 $AA...AA$ 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 density and increase 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 to 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 dataset are moved to appear later in the order.

The number of distinct k-mers associated with a minimizer in a particular dataset 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 smaller load, in order to ensure low 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 AdaOrder (Adaptive Order), iteratively updates the minimizer order based on an estimate of the minimizer loads in the data in order to reduce the maximum load. We demonstrated its ability to lower the maximum load compared to all predefined orders, except the frequency based order.

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 datasets, 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 to Gerbil. Our tests also showed that the orders produced by our method produced superior results when transferred across datasets 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 datasets and species, are publicly available at `github.com/Shamir-Lab/AdaOrder`.

The results of this thesis were accepted for publication in *Journal of Computational Biology* (Flomin et al., 2022).

2 Biological background

2.1 DNA and Genomics

The genetic information of living organisms is coded in DNA (deoxyribonucleic acid) molecules. The DNA molecule is composed of two intertwined chains (strands) of nucleic acids, which forms the well-known double helix structure of the DNA molecule (Crick et al., 1954). The nucleic acids, also called *bases*, that constitute the DNA molecule are cytosine, guanine, adenine or thymine, which are denoted as C, G, A and T, respectively. The bases A and T are complementary to each other, and similarly the bases C and G are complementary. In the two chains of the DNA double helix, matched bases in the two sequences are complementary. In other words, base A (C) in one chain can match (form a bond) only with base T (G) on the other chain, and vice versa. The two chains have opposite orientations, thus each one is reverse-complementary of the other. As a result of the complementarity, the

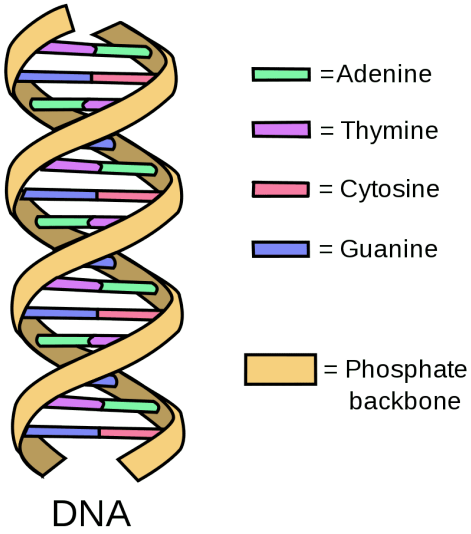


Figure 1: **DNA structure**
Source: https://www.ashg.org/discover-genetics/building-blocks/

sequence of one chain completely determines the other chain. A schematic figure of DNA is shown in **Figure 1**. The genetic information coded in the DNA determines the instructions to produce RNA molecules and proteins, which are building blocks for the functionality of the living organism.

The totality of DNA sequence of an organism is called its *genome*. Genomes vary in scale: The human genome is about 3 billion bases long, and consists of 23 chromosome pairs. The smallest recorded DNA-viral genome *Circovirus SFBeef* has only 859 bases. Most sequenced bacterial genomes are 0.5-5 million bases long. *Genomics* is the study of the genomes of organisms. With the rise of new technologies for analyzing DNA and RNA, the field of genomics has leaped forward with numerous advancements and discoveries (Shendure and Ji, 2008).

2.2 DNA sequencing and next generation sequencing

DNA sequencing is the process of determining the sequence of bases in continuous sub-sequences of an organism's genome. After performing DNA sequencing, using any relevant method, the genomic data produced is used in studies and applications in molecular biology, bioinformatics, genomics, etc. In medicine, genomic sequences of individuals are used for diagnosing genetic diseases, assessing the risk of getting sick with a disease, and predicting the response of patients to treatments (Dewey et al., 2012) etc. In agriculture, sequencing of plants and animals allows the identification of desirable and undesirable traits. By genetic engineering of crops, better, cheaper and safer crops are developed (Purugganan and Jackson, 2021).

Next Generation Sequencing (NGS) refers to novel high throughput sequencing (HTS) technologies that were developed over the last fifteen years. These technologies revolutionized the field of genomics, by allowing to produce massive genomic datasets in a short time and very cheaply (Shendure and Ji, 2008). An older method for sequencing, Sanger sequencing, was used to sequence the first human genome over the course of a decade, while nowadays using NGS technologies it can be done in a single day, and for about one millionth of the cost.

Sequencing technologies produce numerous short contiguous genomic segments, called *reads*. Each read is a subsequence of the genome, possibly with some errors. The location of each read along the genome as well as its strandedness (straight or reverse-complement) are random and unknown. Some NGS technologies allow to sequence short segments (e.g., 50-200 contiguous bases) while other sequence much longer segments (e.g., 10000 contiguous bases) (Goodwin et al., 2016). There are advantages and down-sides to each technology. For instance, technologies that produce short reads are usually less prone to sequencing errors, compared to technologies that produce long reads. On the other hand, long read techniques make assembly much easier (more on this below).

Since NGS is so fast and cheap, it has been adopted very broadly and is now ubiquitous in biology and medical research. In translational medicine, it allows to integrate genomic analysis in clinical practice, and to broaden the diagnosis and

treatment possibilities for patients.

2.3 Genome assembly

Reconstructing the full genome of a species (animal, plant, microbe, virus, etc.) is often a difficult task. With the sequencing technologies available today, one cannot directly sequence the genome in "one shot". The current approaches sequence many reads from the genome, obtained from random unknown locations in it, and together covering the genome multiple times (30-50). Afterwards one needs to reconstruct the full genome from the these reads, in a process akin to putting together a puzzle from its pieces.The high coverage creates overlaps between reads, which are the key to the reconstruction process. This task is called *genome assembly*. A schematic figure of this process is shown in **Figure 2**. Some central assembly algorithms are described in **Section 3.6**.

2.4 k-mer counting

A k-mer in a sequence is a contiguous sub-sequence of k letters in it. k-mer counting is the process of determining how many times each k-mer appeared in an input dataset. **Figure 3** presents a schematic diagram of this process. k-mer counting of sequenced reads is a useful tool for both computational and biological analysis, with numerous applications in biology and medicine. For viruses, it was shown by Alam and Chowdhury (2020) that the k-mer abundance of a sample can accurately classify the type of a RNA virus. For humans, Annalora et al. (2018) showed that the k-mer abundance in cancer cells from the Ewing's family of tumors can help in drug discovery targeting this specific cancer type. In metagenomic analysis, k-mer counting and analysis enables metagenomic classification (Breitwieser et al., 2018). Some central k-mer counting algorithms are described in **Section 3.5**.

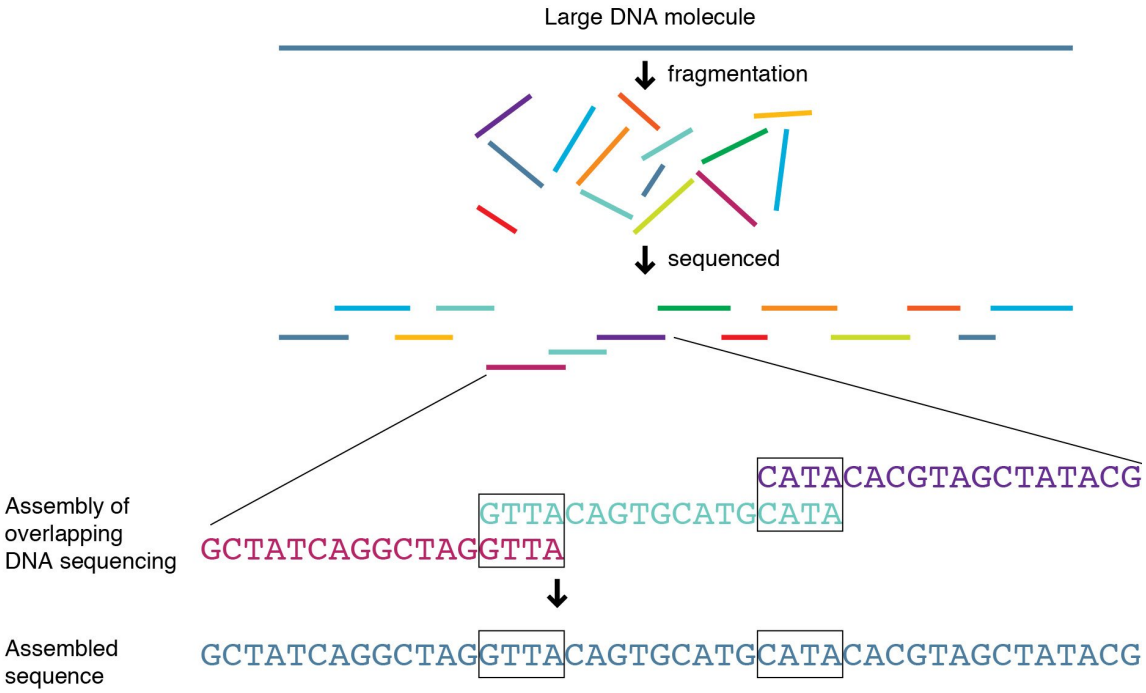


Figure 2: **Genome assembly using overlapping reads**
Source: https://stringfixer.com/files/16274765.jpg

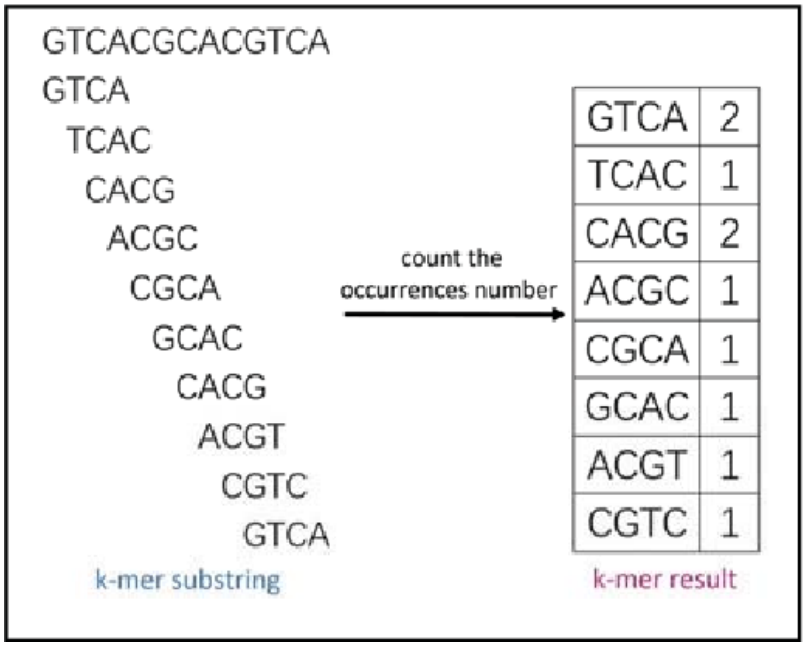


Figure 3: ***k*-mer counting.**
Source: https://www.semanticscholar.org/paper/K-mer-Counting%3A-memory-efficient-strategy%2C-parallel-Xiao-Li/02563dbd80c0a157cbf7202f31be3b2db391457b/figure/0

3 Computational background

In this section we provide background on computational concepts and algorithms used in this study. We start with basic definitions and then describe key relevant algorithms.

3.1 Basic definitions

Natural m-mer mapping: A mapping of m-mers to natural numbers is defined as follows. We treat each m-mer as a number in base 4, with the following DNA base encoding: $A = 0, C = 1, G = 2, T = 3$. For example, with $m = 4$, $AAAA = (0000)_4 = 0$, $TTTT = (3333)_4 = 255$, $ACGT = (0123)_4 = 27$. This mapping is called the *natural mapping* and denoted as $Natural : \Sigma^m \rightarrow \{0, ..., 4^m - 1\}$. We refer to m-mers and values in $\{0, ..., 4^m - 1\}$ interchangeably.

Reverse complement and canonical form: Given an m-mer x, its *reverse complement* is constructed by reversing the order of the bases in x and taking their complements. The complements of bases A, C, G, T, are T, G, C, A respectively. Given an m-mer x, $\bar{x}$ denotes its reverse complement. For example, $\overline{ACCG} = CGGT$.

The *canonical form* of an m-mer x is the smaller of x and $\bar{x}$ with respect to the natural mapping. We denote x's canonical form by $Canonical(x)$. For example, $Canonical(CGGT) = ACCG$.

We refer to the set of all canonical m-mers as $c_m = \{Canonical(x) | x \in \Sigma^m\}$.

m-mer orders: An *order* o on Σ^m is a function $o : \Sigma^m \rightarrow \mathbb{R}$. An order o can be *normalized* to $o_{norm} : \Sigma^m \rightarrow \{0, 1, ..., |\Sigma|^m - 1\}$, with some rule for breaking ties consistently. We further elaborate on m-mer orders in **Section 3.3**.

3.2 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 (Roberts et al., 2004; Schleimer et al., 2003). 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 in the sequence.

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.

Minimizers are used to partition a sequence as follows (Compare **Figure** 4). The sequence is scanned from left to right, and in every window of length k the minimizer m-mer is chosen. The k-mer is assigned to the partition labeled with the minimizer. Since the strandedness of the sequence within the genome is unknown, it is undistibuishable from its reverse-complement, and therefore the canonical mapping is used. Applying a minimizer scheme $f_{o,m,k}$ over a dataset partitions its k-mers into at most $|c_m|$ partitions, where each partition corresponds to a specific minimizer.

Two important metrics in the evaluation of a minimizer scheme and its partitioning quality are particular density and load.

The *particular density* of a sequence or a set of sequences is defined to be the number of selected positions of minimizers divided by the number of possible m-mer positions, using a specific minimizer order. A low particular density implies that a relatively small number of minimzers were chosen. A low particular density is preferable in several applications that use minimizers, e.g., read mapping and reads binning and compression.

Load is a metric used for evaluating the quality of a minimizer order which is mainly relevant to k-mer counters and genome assembly algorithms. The *load* of a minimizer x over a dataset and a fixed k, is the number of distinct k-mers that their minimizer is x. More precisely, given a minimizer scheme $f_{o,m,k}$ and a dataset 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 dataset D to be x's load divided by the total number of distinct k-mers in the dataset. We denote it by $r(x, D)$. The

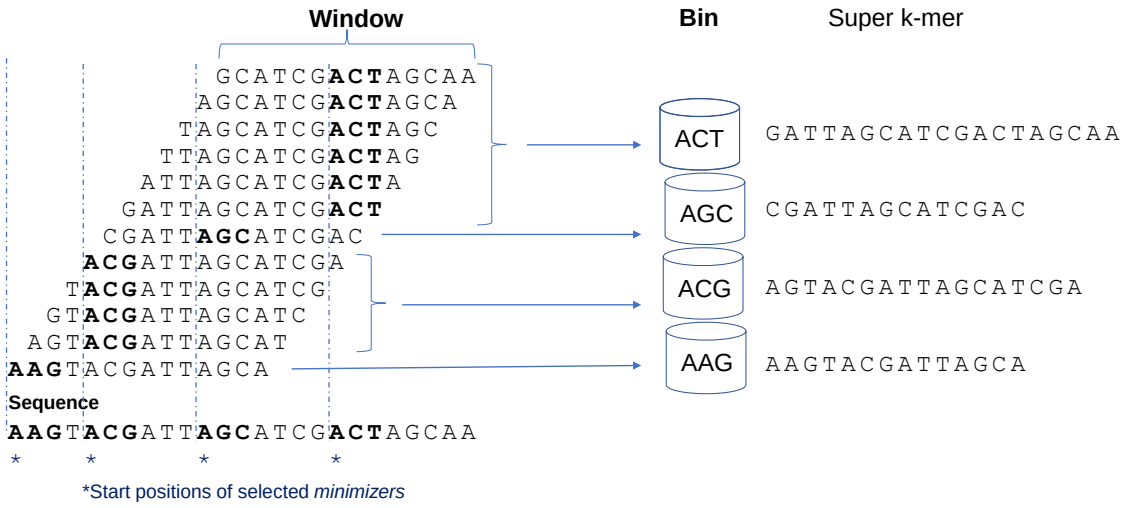


Figure 4: **Illustration of a minimizer scheme and binning.** Here $k = 12$ and $m = 3$. The input sequence is broken into windows of length k, and in each window the m-long minimizer according to the lexicographic order (shown in bold) is selected. The k-mer is assigned to the partition whose label 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.

maximum (relative) load is the (relative) load of the minimizer with the highest (relative) load. Maximum (relative) load is defined analogously for bins. For example, a low maximum load is preferable for an economic memory usage in k-mer counting applications.

Both particular density and load were investigated in recent studies and were shown to provide meaningful measures of the performance of a minimizer scheme within a bioinformatic application (Orenstein et al., 2017; Marçais et al., 2017; Nyström-Persson et al., 2021; Ben-Ari et al., 2021). For example, Ben-Ari et al. (2021) showed that a small particular density and maximum load improves the efficiency of genome assembly using de Bruijn graphs in the MSP algorithm (Li et al., 2013).

3.3 m-mer orders

The order used in minimizer schemes is crucial to its efficiency in k-mer counting and genome assembly algorithms. We will now describe several popular m-mer orders.

We define the *lexicographic order* to be: $o_{lexico}(x) = Natural(x)$.

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.

We define a pseudo-random order by performing a 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:

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

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

The *frequency order* is a minimizer order based on statistics collected from a specific dataset (Chikhi et al., 2015). It orders the m-mers in increasing order according to their frequency in the dataset. For our analysis we defined it as follows: Given a dataset 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.

3.4 Partitions, bins and unevenness

In *minimizer partitioning* each k-mer in the input dataset 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* is the longest substring for which all k-mers have the same minimizer (Li et al., 2013). **Figure 4** shows an example of partitions and super-k-mers.

Unevenness aims to describe how balanced a minimizer scheme is over a particular dataset. Given a minimizer scheme $f_{o,m,k}$ and a dataset 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.

3.5 k-mer counting algorithms

Since k-mer counting is a common process in many bioinformatic applications, many k-mer counter algorithms were developed in recent years. A naive k-mer counting algorithm is shown in **Algorithm 1**.

Current state-of-the-art k-mer counters are Gerbil (Erbert et al., 2017) and KMC3 (Kokot et al., 2017). Gerbil and KMC3 are similar in their first step and different in their second step. First, they both distribute the reads dataset into bins for processing each bin independently. Then, for each bin, KMC3 performs counting using a sorting mechanism, while Gerbil employs a hash table in order to count the k-mer appearances.

Algorithm 1 k-mer counting

Input: A set of reads $D = (S_1, S_1, \ldots, S_M)$, where $|S_i| \geq k$.
Output: A mapping containing a key for each k-mer in D with the number of its appearances in D as value.

```
1: A ← a mapping of k-mers to integers
2: for i from 1 to M do
3:     for each k-mer s in S_i do
4:         if s is a key in A then
5:             A[s] ← A[s] + 1
6:         else
7:             A[s] ← 1
8: return A
```

KMC3 is an improvement of KMC2, which employs the same two stages (Kokot

et al., 2017; Deorowicz et al., 2015). It uses a bin mapping algorithm to achieve more balanced bins, and it uses a better sorting algorithm, which improves runtime and memory usage. The bin mapping algorithm samples the data, estimates the size of each minimizer partition, then packs several minimizers into the same bin while trying to keep the bins balanced. In the first stage, the algorithm distributes the input reads dataset into bins using the signature order on minimizers. Then it processes each bin independently. Each bin's k-mers are sorted and then duplicates are removed while maintaining a counter for each k-mer. This process outputs the k-mer counts of the reads dataset. An outline of the algorithm is described in **Algorithm 2**.

Algorithm 2 KMC3

Input: A set of reads $D = (S_0, S_1, \ldots, S_{M-1})$, where $|S_i| \geq k$. m - minimizer length, B - number of bins.
Output: k-mer counts

1: Distribute k-mers into B bins using the pre-created minimizer-bin mapping
2: **for** b from 1 to B **do**
3: Sort bin b in-memory
4: Count k-mer appearances by removing duplicates
5: Merge the k-mer counts of each of the bins
6: **Return** the unified k-mer count

Gerbil is a memory efficient k-mer counter (Erbert et al., 2017). In a paper that benchmarked several k-mer counters over several datasets, Gerbil outperformed all other methods in peak memory usage (Manekar and Sathe, 2018). Gerbil works in two stages. First it distributes the reads into bins using a minimizer scheme with the signature order. The distribution heuristic aims to create evenly-sized bins, and is described in Section 3.7. Each thread in Gerbil loads reads into memory one-by-one, breaks it into super-k-mers using minimizers with the signature order, yielding super-k-mers that have only one minimizer in them. It stores the k-mers in the bins in a memory-efficient manner as super-k-mers and with a 2-bit encoding for the bases. Each k-mer is distributed to the bin corresponding to its minimizer. Then Gerbil iterates over the bins. For each bin it loads the stored k-mers sequentially, and inserts them into a dedicated hash table for the processed bin. The second step

creates a k-mer counts for each bin. The counts from all the bins are summed to yield the k-mer counts for the entire dataset. An outline of the algorithm is shown in **Algorithm 3**.

Algorithm 3 Gerbil

Input: A set of reads $D = (S_1, S_2, \ldots, S_M)$, where $|S_i| \geq k$. B - number of bins.
Output: k-mer counts

1: Compute minimizer to bin mapping
2: **for** i from 1 to M **do**
3: Break S_i into super-k-mers and put each one in the bin corresponding to its minimizer
4: **for** i from 1 to B **do**
5: Count k-mer appearances in bin i
6: Combine the k-mer counts from each bin
7: Return k-mer counts

3.6 Genome assembly algorithms

Many genome assembly algorithms were developed over the years (Idury and Waterman, 1995; Myers et al., 2000; Zerbino and Birney, 2008; Chikhi et al., 2016; Li et al., 2013; Ben-Ari et al., 2021). Some algorithms perform assembly from scratch (de novo), while others try to assemble the genome via mapping sequences into an existing partially assembled genome. We will focus on de novo genome assemblers.

Older genome assemblers used the *overlap-layout-consensus* (OLC) paradigm in order to perform assembly, while most modern algorithms use de Bruijn graphs or similar variations of it in order to perform it (Li et al., 2011). The latter paradigm is considered to be superior in terms of memory and running time. We will now elaborate on the two different approaches for genome assembly.

The OLC paradigm consists of 3 stages as its name suggests: (i) Overlap; (ii) Layout (iii) Consensus. The overlap step finds overlaps between all reads: For each read is finds all the other reads whose prefix overlaps with its suffix. This action creates a graph with vertices corresponding to reads and directed edges between reads correspond to an overlap relation. The layout step tries to merge adjacent reads in the

overlap graph, to create long contigs, a long string composed of multiple reads. The merging of reads into contigs is a delicate task, since a merge action of two reads must take into account all neighbors of both reads, so the merge action wouldn't contradict the overlap relation with other reads. The consensus step tries to correct sequence errors in contigs. Sequencing errors are common, and therefore this step makes the assembly's result more reliable. Upon finishing the three stages, the long corrected contigs are the result of the genome assembly.

The OLC method requires great computational power, as the overlap graph requires the comparison of all reads with all the other reads. The size of the overlap graph is potentially very large, requiring a large amount of memory and CPU.

We will now outline the de Bruijn graph assembly method. A de Bruijn graph (DBG) is a directed graph data structure with a special restrictions over its vertices and edges. Firstly, we define an alphabet Σ and fix k. Each vertex corresponds to a k-mer over Σ, and each k-mer is represented at most once in the graph. Each edge connects two vertices if and only if the corresponding k-mers overlap in $k-1$ characters. More precisely, $(u, v) \in E$ if and only if the last $k-1$ characters of u are v's $k-1$ first.

When using a de Bruijn graph for assembly, reads are broken into overlapping k-mers, and each k-mer is introduced into the graph. Upon building the graph it is common to compact it by merging long non-branching paths (i.e., a path whose vertices have an in- and out-degree of 1).

The use of de Bruijn graphs in genome assembly was first suggested by Idury and Waterman (1995), and prompted the fast improvement and evolution of genome assembly algorithms. It allowed the design and development of faster and more memory efficient algorithms. Moreover, since de Bruijn graphs have a restricted structure and bounded in-and-out-degrees, it is easier to work with compared to the huge overlap graphs produced by the OLC paradigm.

In recent years, Chikhi et al. (2016) developed BCALM2, and Holley and Melsted (2020) developed Bifrost. These algorithms are considered to be the state algorithms for compacting the de Bruijn graph.

The minimum substring partitioning (MSP) algorithm constructs the de Bruijn

graph for a given reads dataset (Li et al., 2013). MSP consists of three stages: partitioning, mapping and merging. Partitioning is the distribution of the input dataset into bins, using minimizers and hashing of minimizers to bins. The mapping step creates, for each bin independently, a de Bruijn graph of its sequences. The merging step merges these graphs into the final de Bruijn graph of the whole dataset.

BCALM2 is an algorithm for constructing the compacted de Bruijn graph quickly and in low memory (Chikhi et al., 2016). The algorithm's input is the set of all k-mers in the dataset (i.e., it does not work directly on the reads dataset), thus requiring to run a k-mer counter prior to its own run. The use of the most economic k-mer counter prior to running BCALM2 will allow for further reduction in the overall time and memory requirements of the process of creating the compacted de Bruijn graph.

3.7 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 balance the loads of the bins better. We note that 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, minimizers are handled in increasing signature order and the next k-mer is added to the next bin. The order of the bins to which minimizers are added is reversed each time a minimizer has been added to every bin.

KMC3 samples a portion of the dataset in order to estimate the partition size of each minimizer (Kokot et al., 2017). 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 **Algorithm 2**.

4 Methods

We developed an algorithm called AdaOrder that constructs a minimizer order with low maximum load for a given dataset, based on statistics collected on the dataset. 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; (iii) Map minimizers to bins using the estimates from (ii). We integrated this process into Gerbil and call the modified algorithm DGerbil.

4.1 AdaOrder

AdaOrder is a heuristic that aims to produce a minimizer order with low maximum minimizer load in a given dataset. 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 dataset in order to capture a minimizer with a large load. A single round proceeds as follows: (i) Iterate over the dataset'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; (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 rank 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 4** describes AdaOrder.

In our implementation of AdaOrder some optimizations were introduced to **Algorithm 4** for a faster running time. See **Supplementary Section S4** for details.

4.2 Mapping minimizers to bins

We first estimate each partition size through sampling (**Supplementary Algorithm S1**), and then map minimizers to bins using these estimates. The bin mapping algorithm is shown in **Algorithm 5**; it heavily relies on KMC3's implementation.

Algorithm 4 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 $f_{o,m,k}$

The only differences are: (i) counts are not modified, and (ii) k-mers with zero count are spread evenly across the bins.

4.3 DGerbil

DGerbil is a variant of 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, in order to lower the RAM usage. The algorithm for DGerbil is shown in **Algorithm 6**. Gerbil's binning method and counting method are described in **Section 3.5**.

Algorithm 5 BinMapping

Input: Dataset D. M - minimizer length. B - number of bins.
Output: A mapping of the partitions to bins.

Sample D and obtain a count $c(m)$ for each minimizer m
Map minimizers with $c(m) = 0$ evenly across B bins
Sort minimizers with $c(m) \neq 0$ in decreasing order of $c(m)$
$nb = B$; $Tot = \sum_m c(m)$ ▷ #remaining bins; total size of remaining partitions
$m = 1$ ▷ the minimizer with largest size that is still unmapped
for *bin* from 1 to B **do**
    $Mean = Tot/nb$ ▷ mean size of the remaining bins
    $size = 0$ ▷ filled size of the current bin
    while $size < Mean$ **do**
        add $c(m)$ to size, map m to bin *bin* and increase m by 1
    $Tot = Tot - size$; $nb = nb - 1$
Return the mapping of partitions to bins and the filled bin sizes

Algorithm 6 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

Generate minimizer ordering, *order*, using AdaOrder initialized with o_{sig} (**Algorithm 4**)
Collect minimizer partition sizes, *stats*, using MinimizerStats (**Algorithm S1**) and *order*
Create a mapping of minimizers to bins, *mapping*, using BinMapping (**Algorithm 5**) and *stats*
Apply Gerbil's binning method over D using *order* and *mapping*
Use Gerbil's counting method to count k-mers in each bin
Return k-mer counts

4.4 FGerbil

For the sake of comparison, we also implemented a variant of Gerbil that uses frequency order, called FGerbil. It first counts how many times each m-mer appeared in the dataset, and orders m-mers by their frequency. It then uses the resulting

frequency order within Gerbil, and performs bin mapping as DGerbil does.

4.5 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.

Dataset	Size (10^9 bytes)	Avg. read length	Species
HS1	292	151	H. sapiens
HS2	347	202	H. sapiens
AT	72.7	4804	A. Thaliana
FW	406	300	Fresh water metagenome
FV	14.1	508	F. vesca
NC	45.9	7778	N. crassa
DM	10.7	152	D. melanogaster
MB	198	101	M. balbisiana

Table 1: **Characteristics of the benchmark datasets.** Note that the 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.

5 Results

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

We used eight datasets; seven were those used in a recent k-mer counter benchmark (Manekar and Sathe, 2018); the eighth was a large freshwater metagenomic dataset (FW (Mehrshad et al., 2018), SRR6787039). See **Table 1** for the properties of the datasets. Both AT and NC are long reads datasets 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 dataset. Minimizer length was always $m = 7$.

5.1 AdaOrder reduces maximum load and unevenness

We applied the lexicographic, signature, random, frequency and AdaOrder orders on four large datasets for $k = 28$ and 55. We used the same values of k as in Manekar

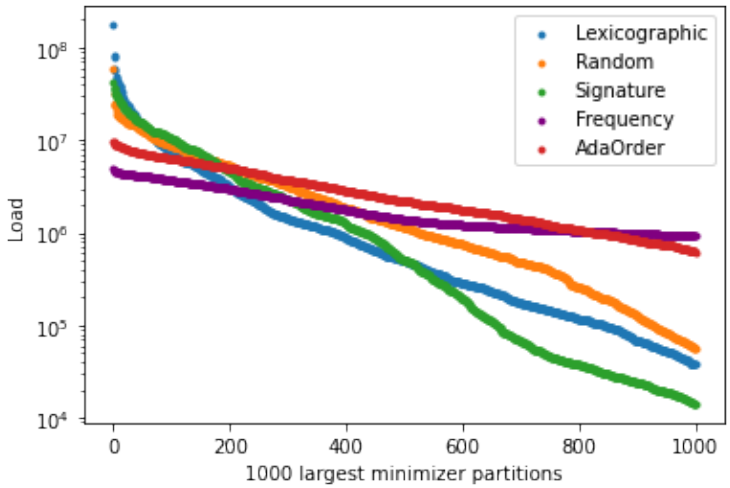
Dataset	k	Lexicographic	Random	Signature	AdaOrder	Frequency
HS1	28	78.5	20.1	14.6	5.7	**3.0**
	55	157.1	51.7	36.7	10.2	**4.8**
HS2	28	83.7	22.1	15.6	5.8	**3.1**
	55	175.3	60.0	41.7	9.9	**5.1**
FW	28	58.1	54.0	27.1	11.0	**5.3**
	55	112.3	70.2	58.8	19.0	**8.6**
AT	28	35.1	4.7	12.0	0.6	**0.5**
	55	6.0	1.2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

Table 2: **Maximum load** $\times 10^{-6}$

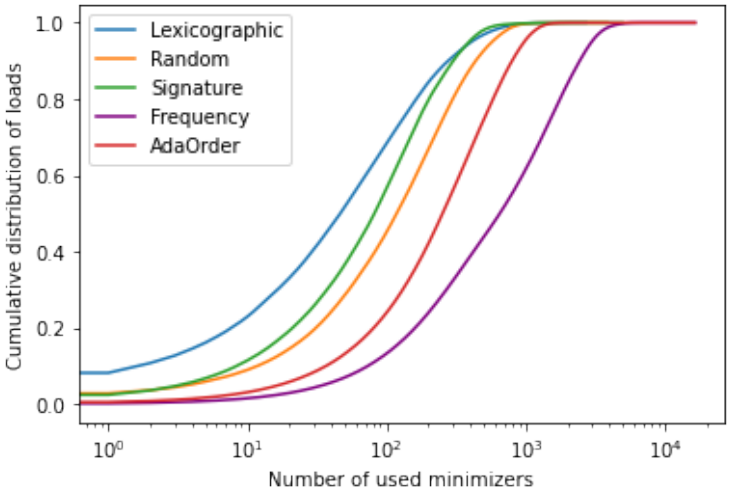
and Sathe (2018). In the random order we used a different random mask for each generated order. Maximum load and unevenness results are shown in **Table 2** and **Table 3**, respectively. Frequency order consistently had the lowest maximum load and unevenness followed by AdaOrder, while the predefined orders were substantially worse.

We also calculated the distribution of load across minimizers for each order. In **Figure 5a** 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 **Table 2** and **Table 3**. **Figure 5b** shows the cumulative distribution of the load for all minimizers. AdaOrder and frequency order did a good job in balancing the top loads compared to 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.

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.



(a)



(b)

Figure 5: **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.

Dataset	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**

Table 3: **Unevenness** $\times 10^7$

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

Figure 6a 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. The trends are similar for sample size $100,000$ and $1,000,000$.

Figure 6b shows the effect of varying N, the sample size per round. Both maximum load and unevenness decrease consistently until $N = 100,000.$, with smaller and less clear decrease for larger N. This demonstrates the benefit of choosing a large sample.

Figure 6c shows the effect of varying p, the penalty factor. The load decreases first as p increases, but no clear trend is observed for $p \geq 0.1$, where substantial jumps are observed in the load. A similar trend but with lower variability is observed for unevenness. The trends are similar when using 5000 and 10000 rounds. The high variability for high values of p is likely due to large changes in the order in the final rounds after all the top loads have already been made relatively even. **Supplementary Figure S1** shows a similar analysis for $N = 10^4 - 10^8$ and different number of rounds.

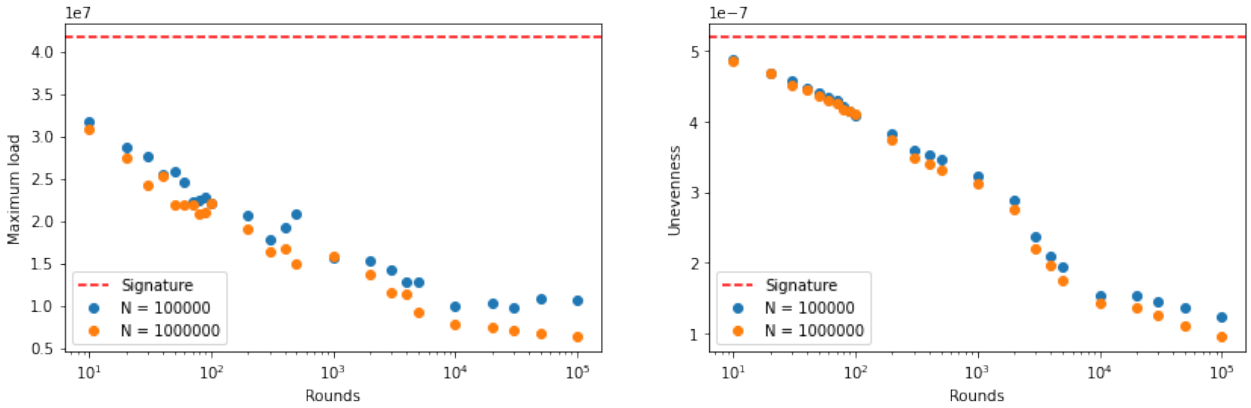
Based on the analysis above and the theoretical analysis in **Supplementary Section S1** we established recommended parameters for running AdaOrder. See **Supplementary Section S3** for details.

5.3 Transferring the order across datasets

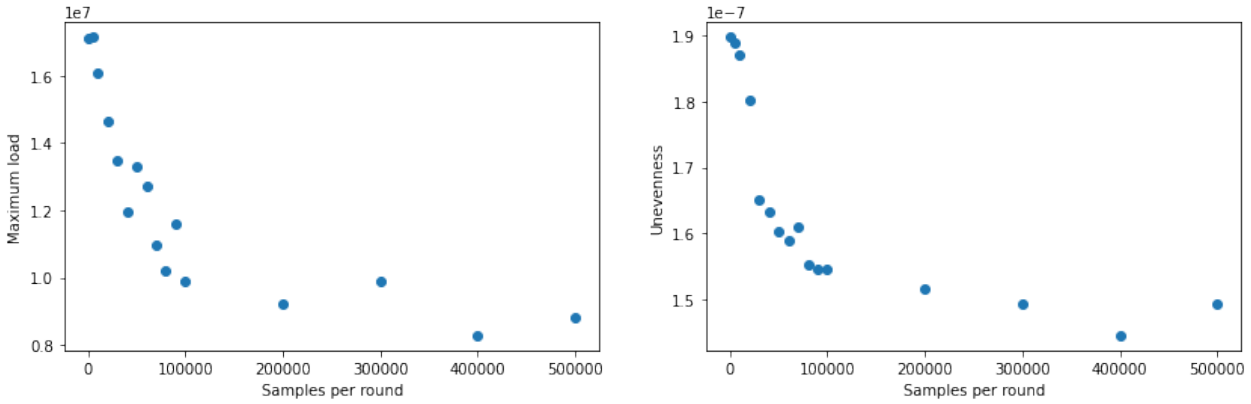
We wished to see whether an order produced on one dataset can assist in creating good orders for other datasets from the same species, either by running it as is, or by using it as a starting order in the optimization for the other datasets. We reasoned that if this is the case, then it could speed up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

Suppose we have a source dataset *src* and a destination dataset *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*.

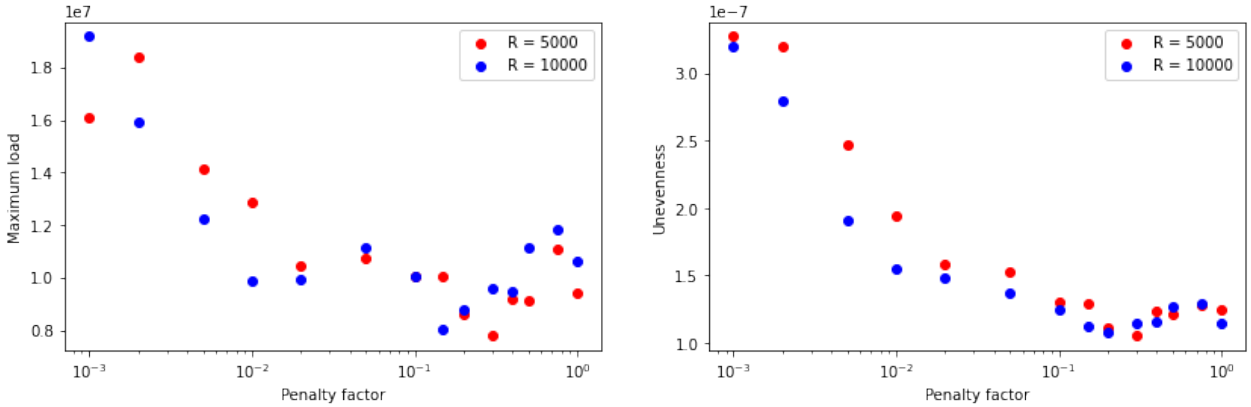
The results are shown in **Figure 7** and **Figure 8**. We see that transferring the order works similarly to the AdaOrder results produced for the destination dataset independently of *src* (the dashed green line). Moreover, starting from the *src* order, the algorithm can improve unevenness and maximum load within far fewer optimization rounds compared to running AdaOrder for *dst* from scratch (< 1000 vs. 10^4). We conclude that the order can be transferred across datasets from the same species,



(a)



(b)



(c)

Figure 6: **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 dashed red 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.

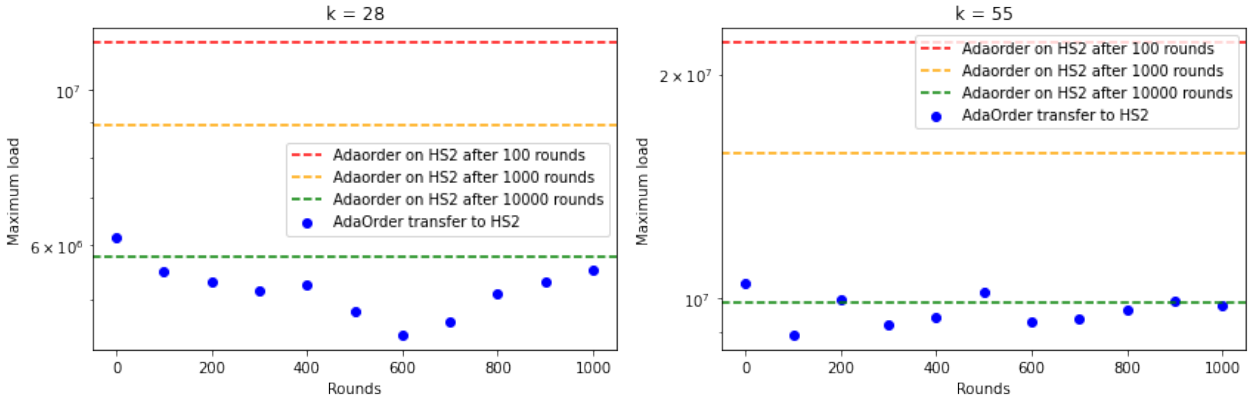
saving most of the running time and preserving the order quality.

5.4 DGerbil reduces memory usage

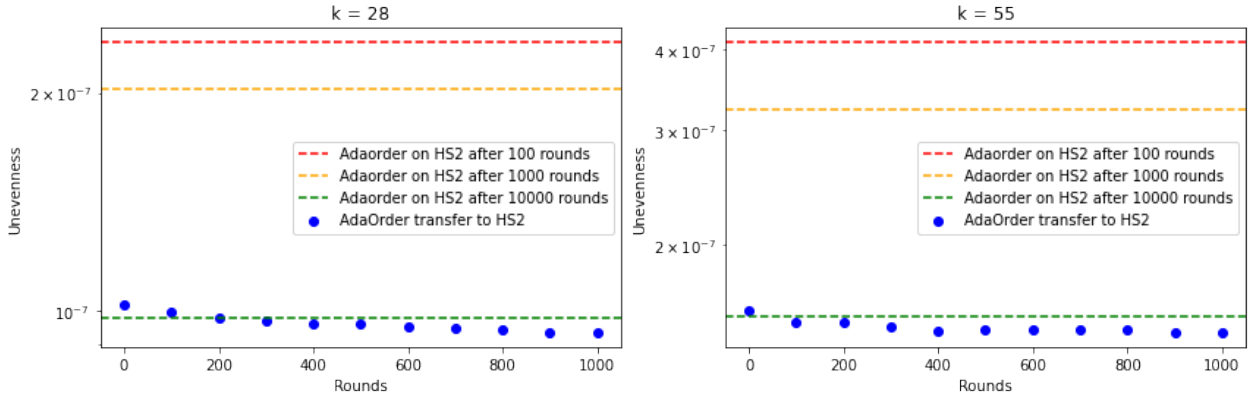
We compared the performance of Gerbil, DGerbil and FGerbil. All algorithms were run with 512 bins and $k = 28, 55, 70, 90$ on each dataset (28 and 55 were used in Manekar and Sathe (2018)). All the experiments were measured on a 128-core server with 64 3.35 GHz CPUs and 1000GB of RAM (AMD EPYC 7702). 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 IO operations were performed against it (e.g., reading datasets 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.

The results for the four datasets with the highest RAM usage are shown in **Figure 9**. The results for the other four datasets are in **Supplementary Figure S2**. The figures show the runtime and RAM usage on each dataset. For DGerbil the time of AdaOrder is shown separately. Similarly, for FGerbil, the time of computing the frequency order is shown separately. DGerbil used consistently lower RAM except for the smallest value of k (28), where Gerbil was more memory-efficient. All tools tended to use more memory as k increased. Similar RAM trends were observed for the other datasets (**Supplementary Figure S2**) though absolute numbers were lower.

In terms of time, it took an average of 6m 50s to run AdaOrder and to collect statistics for binning. This runtime was roughly constant as the sampling process does not depend on the dataset size. For frequency it took on average about 39 minutes to finish collecting statistics for the order and for the binning. For the smaller datasets 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.

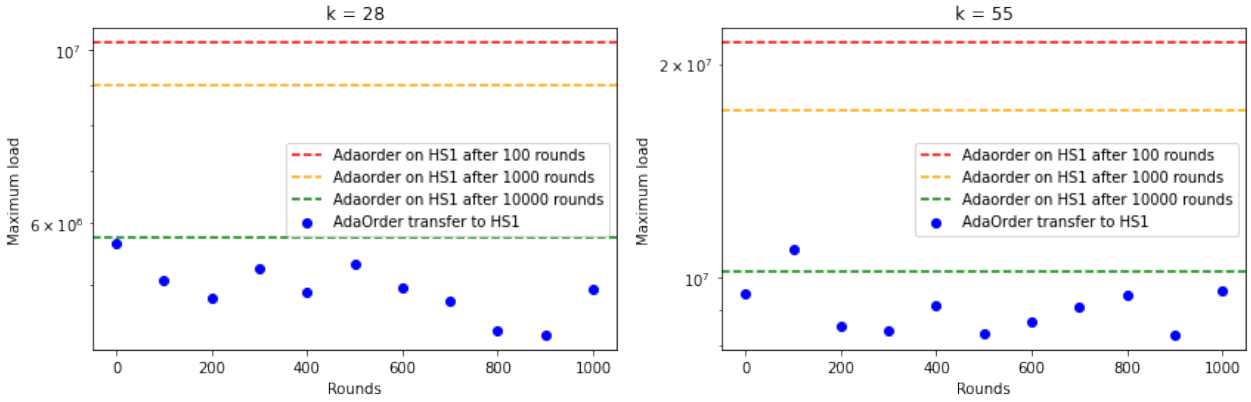


(a) Maximum load

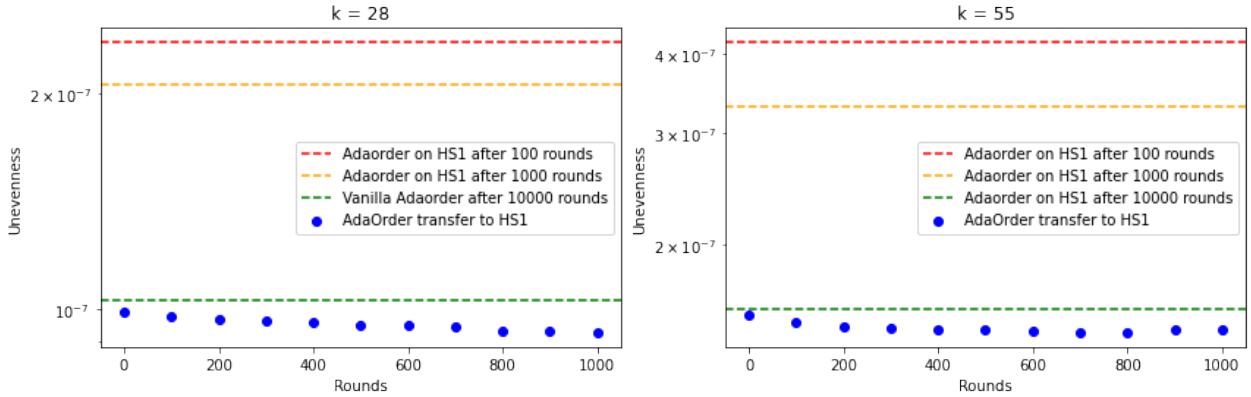


(b) Unevenness

Figure 7: **Order transfer from HS1 to HS2.** 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.

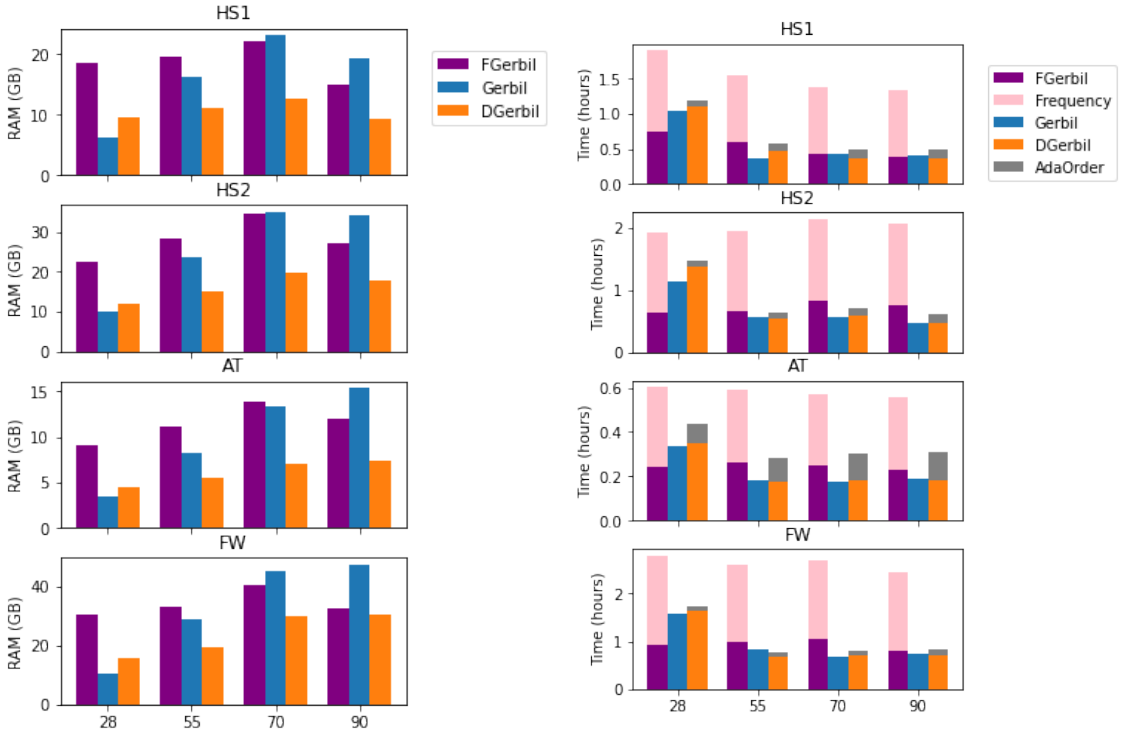


(a) Maximum load



(b) Unevenness

Figure 8: **Order transfer from HS2 to HS1.** 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.



(a) RAM usage

(b) Runtime

Figure 9: **Performance of Gerbil, FGerbil and DGerbil.** (a) RAM usage. (b) Runtime. For DGerbil and FGerbil, AdaOrder and Frequency times for creating the order and collecting binning statistics are shown separately.

6 Discussion

We developed AdaOrder, an algorithm for creating an improved minimizer order on a given dataset by repeatedly adjusting the order to reduce the maximum load. AdaOrder significantly reduced maximum load and unevenness compared to the signature order and to other predefined orders. The load and unevenness of the frequency order were even better, but they 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 in order to map minimizers into bins. This reduced memory usage of Gerbil by 30-50% for medium and large k.

AdaOrder has a roughly constant runtime of under seven minutes regardless of the size of the dataset. For small datasets 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 datasets, 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.

DGerbil outperformed Gerbil in memory, but required a bit more time. We demonstrated that by transferring an order precomputed by AdaOrder on another dataset 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 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. First, can one improve AdaOrder 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). (iv) Is maximum load the best metric to measure the performance of a minimizer scheme when optimizing an order to a dataset? We know that without binning of partitions, the maximum load upper bounds the peak memory usage in hash table based k-mer counters, and therefore is near optimal. But is maximum load efficient in pointing which minimizer should be punished in AdaOrder? Is there a better metric to use in the iterative punishment of AdaOrder? Together, some or all of these ideas could improve the algorithm further.

Second, most existing methods that employ minimizers and orders to partition datasets in k-mer counting and genome assembly algorithms are not based on the properties of the dataset of interest. As far as we know, only Chikhi et al. (2015) with the frequency order and Nyström-Persson et al. (2021) with the universal frequency order, have tried to improve the efficiency of the minimizer algorithm using statistics of the target dataset. Both obtained superior performance in comparison to existing predefined orders. Our approach using AdaOrder had reaffirmed that sequence target-based orders outperform predefined orders. Unlike the previous studies, we

used an iterative optimization of the order and did not create it in one pass over the dataset. Both the iterative and the one pass approach have caveats and are not optimal with respect maximum load. Further research on minimizers and orders is needed to uncover better and perhaps even optimal methods for creating a minimizer order. As all sequence target-based methods for creating datasets are heuristic and do not have a rigorous mathematical basis for their efficiency, further research into the problem of optimizing a minimizer order is required.

Of the two sequence target-based orders we tested, the frequency order achieved superior maximum load and unevenness results compared to AdaOrder. However, its use in Gerbil required more RAM compared to DGerbil. This raises another question - can we design a new k-mer counter, whose performance matches the performance of a minimizer scheme it uses? Designing such a k-mer counter may be the missing piece in achieving a very time and memory efficient application. Leveraging frequency's potential in such a k-mer counter may lead to extraordinary results.

In summary, 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 potential to reduce the memory footprint of other sequence analysis algorithms on large datasets.

References

M. N. U. Alam and U. F. Chowdhury. Short k-mer abundance profiles yield robust machine learning features and accurate classifiers for RNA viruses. *PLOS ONE*, 15(9):1–23, 09 2020. doi: 10.1371/journal.pone.0239381. URL https://doi.org/10.1371/journal.pone.0239381.

A. J. Annalora, S. O'Neil, J. D. Bushman, J. E. Summerton, C. B. Marcus, and P. L. Iversen. A k-mer based transcriptomics approach for antisense drug discovery targeting the ewing's family of tumors. *Oncotarget*, 9:30568 – 30586, 2018.

Y. Ben-Ari, D. Flomin, L. Pu, Y. Orenstein, and R. Shamir. Improving the efficiency of de Bruijn graph construction using compact universal hitting sets. In *Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, BCB '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384506. doi: 10.1145/3459930.3469520. URL https://doi.org/10.1145/3459930.3469520.

G. Benson and M. S. Waterman. A method for fast database search for all k - nucleotide repeats. *Nucleic Acids Research*, 22(22):4828–4836, 11 1994. ISSN 0305-1048. doi: 10.1093/nar/22.22.4828. URL https://doi.org/10.1093/nar/22.22.4828.

F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. Krakenuniq: confident and fast metagenomics classification using unique k-mer counts. *Genome Biology*, 19(1): 198, Nov 2018. ISSN 1474-760X. doi: 10.1186/s13059-018-1568-0. URL https://doi.org/10.1186/s13059-018-1568-0.

H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988. doi: 10.1137/0148063. URL https://doi.org/10.1137/0148063.

R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the

representation of de Bruijn graphs. *Journal of Computational Biology*, 22(5):336–352, 2015.

R. Chikhi, A. Limasset, and P. Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 06 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw279. URL https://doi.org/10.1093/bioinformatics/btw279.

F. H. C. Crick, J. D. Watson, and W. L. Bragg. The complementary structure of deoxyribonucleic acid. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 223(1152):80–96, 1954. doi: 10.1098/rspa.1954.0101. URL https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1954.0101.

S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

F. Dewey, S. Pan, M. Wheeler, S. Quake, and E. Ashley. DNA sequencing: Clinical applications of new DNA sequencing technologies. *Circulation*, 125:931–44, 02 2012. doi: 10.1161/CIRCULATIONAHA.110.972828.

M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms for Molecular Biology*, 12 (1):9, Mar 2017. ISSN 1748-7188. doi: 10.1186/s13015-017-0097-9. URL https://doi.org/10.1186/s13015-017-0097-9.

D. Flomin, D. Pellow, and R. Shamir. Dataset-adaptive minimizer order reduces memory usage in k-mer counting. To appear in J. Computational Biology, 2022.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition, 1979. ISBN 0716710455. URL http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455.

S. Goodwin, J. Mcpherson, and W. McCombie. Coming of age: Ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, June 2016. ISSN 1471-0056. doi: 10.1038/nrg.2016.49.

G. Holley and P. Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21:249, 09 2020. doi: 10.1186/s13059-020-02135-8.

R. M. Idury and M. S. Waterman. A new algorithm for DNA sequence assembly. *Journal of computational biology*, 2(2):291–306, 1995.

C. Jain, A. Rhie, H. Zhang, C. Chu, B. P. Walenz, S. Koren, and A. M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinformatics*, 36 (Supplement 1):i111–i118, 07 2020. ISSN 1367-4803. doi: 10.1093/bioinformatics/btaa435. URL https://doi.org/10.1093/bioinformatics/btaa435.

M. Kokot, M. Długosz, and S. Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 05 2017. ISSN 1367-4803. doi: 10.1093/bioinformatics/btx304. URL https://doi.org/10.1093/bioinformatics/btx304.

H. Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34 (18):3094–3100, 05 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty191. URL https://doi.org/10.1093/bioinformatics/bty191.

Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri. Memory efficient minimum substring partitioning. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2013.

Y. Li et al. MSPKmerCounter: a fast and memory efficient approach for k-mer counting. *arXiv preprint arXiv:1505.06550*, 2015.

Z. Li, Y. Chen, D. Mu, J. Yuan, Y. Shi, H. Zhang, J. Gan, N. Li, X. Hu, B. Liu, B. Yang, and W. Fan. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-Bruijn-graph. *Briefings in Functional Genomics*,

11(1):25–37, 12 2011. ISSN 2041-2649. doi: 10.1093/bfgp/elr035. URL https://doi.org/10.1093/bfgp/elr035.

S. C. Manekar and S. R. Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. ISSN 2047-217X. doi: 10.1093/gigascience/giy125. URL https://doi.org/10.1093/gigascience/giy125. giy125.

G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford. Improving the performance of minimizers and winnowing schemes. *Bioinformatics*, 33(14):i110–i117, 2017.

M. Mehrshad, M. M. Salcher, Y. Okazaki, S.-i. Nakano, K. Šimek, A.-S. Andrei, and R. Ghai. Hidden in plain sight—highly abundant and diverse planktonic freshwater chloroflexi. *Microbiome*, 6(1):1–13, 2018.

E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. J. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H.-H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M. Rubin, M. D. Adams, and J. C. Venter. A whole-genome assembly of ¡i¿drosophila¡/i¿. *Science*, 287(5461):2196–2204, 2000. doi: 10.1126/science.287.5461.2196. URL https://www.science.org/doi/abs/10.1126/science.287.5461.2196.

J. Nyström-Persson, G. Keeble-Gagnère, and N. Zawad. Compact and evenly distributed k-mer binning for genomic sequences. *Bioinformatics*, 03 2021. ISSN 1367-4803. doi: 10.1093/bioinformatics/btab156. URL https://doi.org/10.1093/bioinformatics/btab156. btab156.

Y. Orenstein, D. Pellow, G. Marçais, R. Shamir, and C. Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Computational Biology*, 13(10):e1005777, 2017.

M. D. Purugganan and S. A. Jackson. Advancing crop genomics from lab to field. *Nature Genetics*, 53(5):595–601, May 2021. ISSN 1546-1718. doi: 10.1038/s41588-021-00866-3. URL https://doi.org/10.1038/s41588-021-00866-3.

M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18): 3363–3369, 2004.

S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International conference on Management of data*, pages 76–85. ACM, 2003.

J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26 (10):1135–1145, Oct 2008. ISSN 1546-1696. doi: 10.1038/nbt1486. URL https://doi.org/10.1038/nbt1486.

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

D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Supplementary information

S1 Accuracy of sampling-based load estimation

A key parameter in AdaOrder is N, the number of k-mers it samples in each round. If N is too small, then the penalized minimizer may not actually have a true high load. If it samples too many k-mers then the output minimizer scheme may be superior, but RAM usage and running time will increase. Here we provide a theoretical analysis to guide the choice of N so as to balance AdaOrder's resource usage and the performance of the output minimizer scheme.

We bound the probability of deviation of an estimated load from its true value as a function of N as follows. Let x be a minimizer with true relative load r on dataset D. Let $S \subseteq D$ be a subset with N sampled k-mers. Since k is long and N is relatively small compared to the number of possible k-mers, we make the assumption that the sampled k-mers are distinct. Assume that the sampling process is binomial, namely, x has probability r to be the minimizer of a k-mer. Then for a large enough N we expect x's sample load to be

$$l(x, S) \approx r \cdot N$$

We are interested in choosing N so that the difference from the true relative load is less than δ. The probability of that event is

$$P\left[(1-\delta) \cdot N \cdot r \leq l(x, S) \leq (1+\delta) \cdot N \cdot r\right] \geq \sum_{j=\lceil (1-\delta)\cdot N \cdot r \rceil}^{\lfloor (1+\delta)\cdot N \cdot r \rfloor} \binom{N}{j} \cdot r^j \cdot (1-r)^{N-j}$$

The expression on the left describes the chance that a sampled load differs by at most a factor of δ from the expected sampled load. We lower-bounded this probability by the binomial sum on the right, denoted as $\rho(N, r, \delta)$. This bound can guide us in choosing a value for N, by computing it for typical r values, and a small enough value for δ.

Our empirical results showed that for $m = 7$, maximum load m-mers have rel-

r	0.05	0.01		0.002	
N	10000	10000	50000	50000	100000
$\rho(N, r, \delta)$	0.97	0.68	0.97	0.68	0.84

Table S1: **Sampling estimation accuracy for different values N, r and for $\delta = 0.1$.**

ative load of $\sim 0.01 - 0.06$ for pre-defined orders used in practice (e.g., signature or lexicographic). For example, in human dataset HS2 with $k = 55$ the maximum load was 0.056 for lexicographic and 0.013 for signature. **Table S1** provides values for $\rho(N, r, \delta)$ using different combinations of N and r, with a fixed $\delta = 0.1$. We see that $N = 50000 - 100000$ gives a high probability to estimate the load of m-mers accurately for realistic loads. (For $m > 7$ we expect the maximum relative load to be smaller, requiring larger values of N.)

The above bound is for the load estimate of a single minimizer, but is not necessarily accurate for the *maximum* load. How good the estimate of the maximum load minimizer is depends on how close the second highest load is: if it is close to the maximum, then the identified maximum load minimizer may be incorrect, while if it is far from the maximum, then we can likely estimate the correct maximum minimizer with high accuracy.

In AdaOrder, by design, the top loads decrease and get closer to each other as the iterations progress. We therefore want to have a low probability for a minimizer with a small relative load to end up with the largest count when sampling in an iteration. We do not necessarily expect AdaOrder to detect the exact minimizer with the highest load, but rather a minimizer with a high relative load (e.g., with one of the n highest relative loads). Therefore we wish to bound the probability that a minimizer with a low true relative load $\hat{r}$ that is smaller than the top n relative loads will mistakenly have an estimated sample load X larger than all of the top n loads.

Assume that the relative loads for the top n minimizers are $r_1 \geq r_2 \geq ... \geq r_n$, with corresponding sample loads $X_1, ..., X_n$. Assuming the estimates are independent

(since N is very large), we get:

$$\xi = P(X \geq X_1 \wedge X \geq X_2 \wedge ... \wedge X \geq X_n) = \prod_{i=1}^{n} P(X \geq X_i) \leq \quad (1)$$

$$\prod_{i=1}^{n} P(X \geq X_n) = P(X \geq X_n)^n \quad (2)$$

We evaluated this expression for $n = 10$, using the loads observed in the last round of AdaOrder in the HS2 dataset with $k = 55$: The maximum load was 0.00318, and the 10th largest load was 0.00279. Suppose $\hat{r}$ equals 80% of the maximum relative load in our example (0.00254). We empirically evaluated $P(X \geq X_n)$ by sampling 10^5 times from the binomial distribution with parameters 0.00279 and $N = 10^5$ trials, obtaining $P(X \geq X_n) = 0.136$ and

$$\xi \leq P(X_{\hat{r}} \geq X_{10})^{10} \approx 0.136^{10} = 2.29 \cdot 10^{-9}$$

This calculation suggests that for $N = 10^5$ there is a very low chance of a gross mistake in the identification of a minimizer with a high load.

In **Supplementary Section S2** we show the effect of varying the number of samples, N, in different rounds of AdaOrder, demonstrating the effect described.

S2 The Effect of sampling depth on accuracy of load estimation

We tested how the sampling depth and the number of rounds affect AdaOrder. We used dataset HS2 with $p = 0.01$. **Supplementary Figure S1** shows the relative load as a function of the sample size, for different rounds of the order optimization process. Namely, for each N, the minimizer that would have been selected if sampling stopped after N k-mers, and its relative load, are shown. Results are shown for the first, 1000th and 10^4th round. We see that in the first round the identity and relative load of the maximum load minimizer could be estimated with only $2 \cdot 10^4$ samples. However, in later rounds, as the loads get more even, the load of the chosen minimizer

improves even after 10^5 samples, and it takes even more samples for the identity of the minimizer to stabilize.

S3 Recommended parameters for AdaOrder

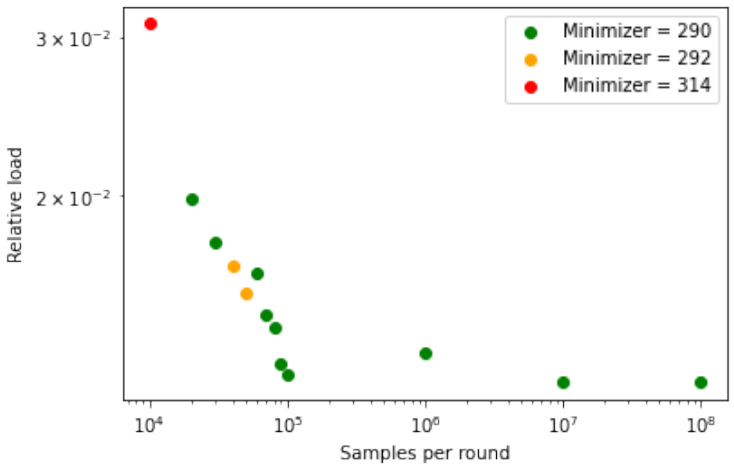
We selected the default parameters of $R = 10^4$, $N = 10^5$, $p = 0.01$ for AdaOrder. These parameters were chosen to limit the runtime of AdaOrder while still achieving a good ordering with low maximum loads (and potentially low RAM usage in DGerbil).

According to **Figure 6a**, increasing R above 10^4 had little effect on the maximum load. The same figure also suggests that $N = 10^5$ did not have a notably worse maximum load than $N = 10^6$. Lower values of N performed worse, as shown in **Figures 6b** and **S1**. In fact, **Supplementary Figure S1** seems to indicate that more samples are needed in the later rounds compared to the initial rounds, since the maximum relative load is getting smaller as AdaOrder progresses (see **Supplementary Table S1**). $N = 10^5$ thus gives a good balance between runtime and maximum load.

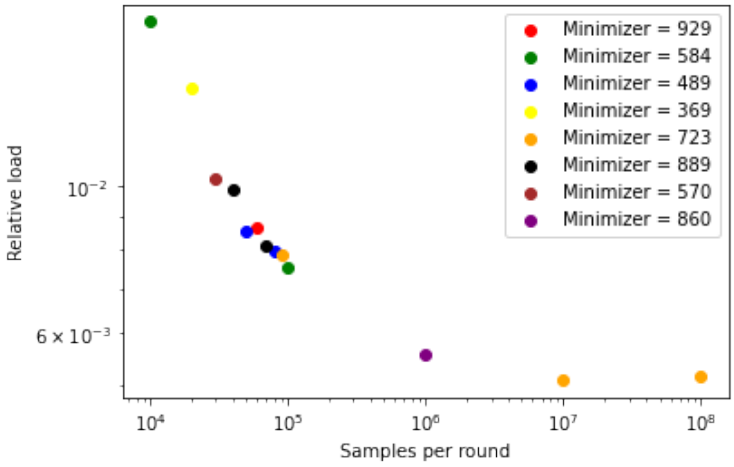
The penalty factor $p = 0.01$ is a conservative choice. Values smaller than 0.01 gave high load and unevenness. Values greater than 0.1 gave unstable load and unevenness. Alternatively, one can choose $0.01 < p \leq 0.1$, in combination with lower R, as **Figure 6c** suggests that somewhat higher p can still lead to more efficient flattening of the order, and thus require fewer rounds.

S4 Implementation details

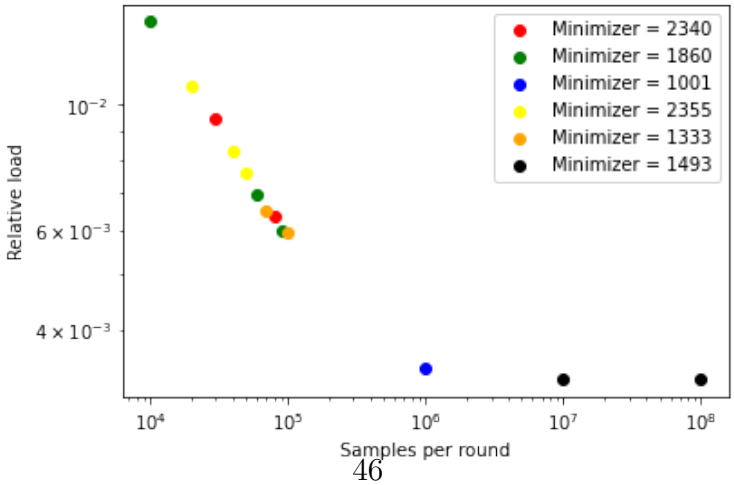
In **Algorithm 4** (AdaOrder), we write on line 8 that we assign x to be the o-minimum m-mer of $S_{read}(i)$. Finding the minimum can be found by scanning the entire string, but instead, by saving the index of the last o-minimum m-mer, in most cases we can just check whether the new m-mer (the last one in $S_{read}(i)$) is o-smaller than the last o-minimum m-mer we found. This optimization affects only the running time of AdaOrder.



(a)



(b)



(c)

Figure S1: **Effect of the sample size and round number on the estimate of maximum load minimizer.** (a) 1st round. (b) 1000th round. (c) 10^4th round. Each dot shows the minimizer to be selected based on the corresponding sample size. The y-axis is the relative load of the selected minimizer. The numbers assigned as minimizer labels are irrelevant, but the same color indicates a repeated minimizer.

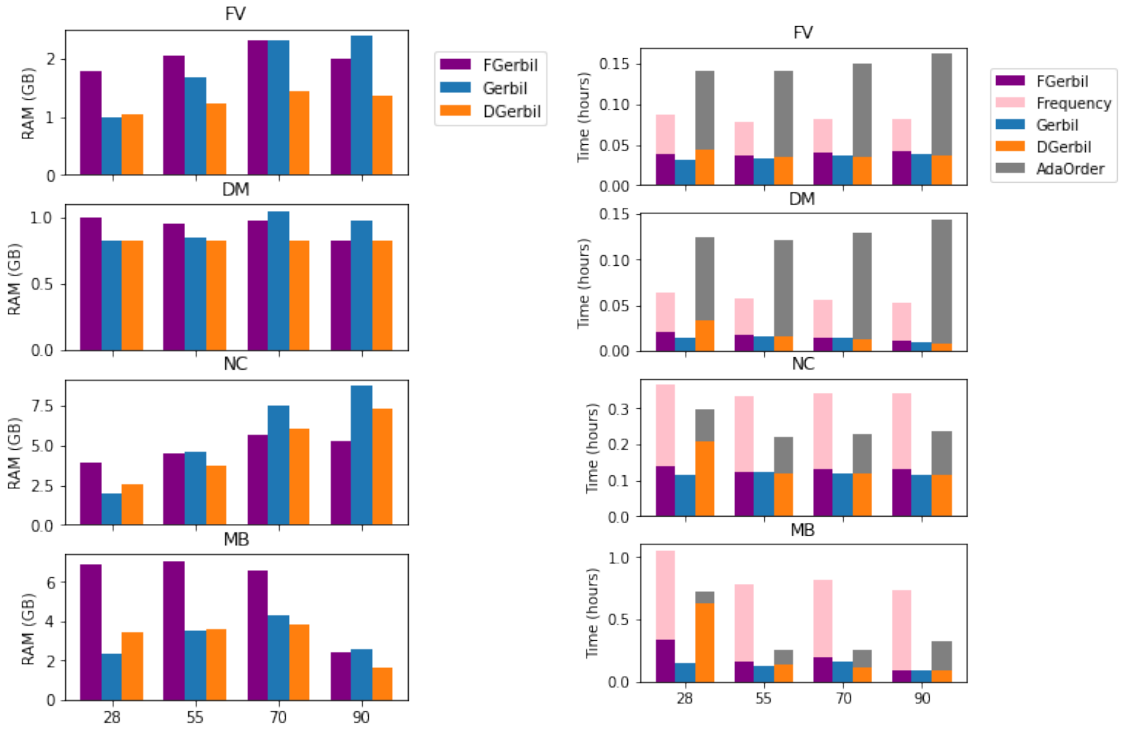
S5 Additional methods and results

Algorithm S1 MinimizerStats

Input: A set of reads $D = (S_0, S_1, \ldots, S_{M-1})$, where $|S_i| \geq k$.
A minimizer scheme $f_{o,m,k}$. E - number of k-mers to sample.

Output: Array of length 4^m with the total size of the super-k-mers in the sample for each m-mer.

$read \leftarrow 0$
$sampled \leftarrow 0$
$A \leftarrow$ zeroed array of length 4^m
while $sampled < E$ **do**
    $minMmer \leftarrow$ the o-minimum m-mer of $S_{read}(0)$
    $A[minMmer] \leftarrow A[minMmer] + k$
    $sampled \leftarrow sampled + 1$
    $i \leftarrow 1$
    while $i \leq |S_{read}| - k$ **do**
        $lastMmer \leftarrow$ the m-mer that starts at $k - m$ in $S_{read}(i)$
        $lastMmerIndex \leftarrow k - m + i$
        if start index of $minMmer < i$ **then**
            $minMmer \leftarrow$ the o-minimum m-mer of $S_{read}(i)$
            $A[minMmer] \leftarrow A[minMmer] + k$
        else if $lastMmer$ is o-smaller than $minMmer$ **then**
            $minMmer \leftarrow lastMmer$
            $A[minMmer] \leftarrow A[minMmer] + k$
        else $A[minMmer] \leftarrow A[minMmer] + 1$
        $sampled \leftarrow sampled + 1$
        $i \leftarrow i + 1$
    $read \leftarrow read + 1$
return A



(a) RAM usage

(b) Runtime

Figure S2: **Performance of Gerbil, FGerbil and DGerbil.** (a) RAM usage. (b) Time. In (b), for DGerbil and FGerbil, AdaOrder and Frequency times for creating the order and collecting binning statistics are shown separately.

תקציר

בבעיית ספירת k-יות (k-mers) נתונים קלט שהוא אוסף רצפי דנא וקבוע k, ויש לספור את מספר ההופעות של כל k-יה בקלט. זהו תהליך נפוץ מאוד בניתוחים רבים של רצפים. הוא ממומש על ידי פיצול הרצפים למקטעים בתאים (bins) שונים כך שתתאפשר ספירה יעילה בנפרד בכל תא ואחר כך איחוד הספירות. הפיצול לתאים מבוסס על שימוש בסדר של כל ה-m-יות האפשריות שנקרא סדר מינימייזרים (minimizer order). לכל מקטע באורך k ברצף, ה-m-יה שבו שהיא בעלת המיקום הגבוה ביותר בסדר נקראת המינימייזר של המקטע והיא קובעת לאיזה תא ישוייך המקטע. מספר רב של אלגוריתמים פותחו לבעייה בשל חשיבותה והמשאבים הרבים שהיא דורשת, ולכל אלגוריתם סדר מינימייזרים משלו.

מטרת המחקר שלנו היתה לייעל את צריכת הזיכרון בתהליך ספירת k-יות. פיתחנו אלגוריתם היוריסטי שנקרא AdaOrder, שמייצר סדר מינימייזרים עבור קלט נתון. העומס של מינימייזר מוגדר כמספר ה-k-יות השונות המשוייכות לאותו מינימייזר. האלגוריתם מעדכן איטרטיבית את הסדר במטרה להוריד את העומס המקסימלי של מינימייזר כלשהו. בכל איטרציה האלגוריתם מחפש את המינימייזר עם העומס הגדול ביותר כרגע, ומשנה את המיקום שלו בסדר כך שיהיה פחות מתועדף. בכך אנו מגדילים את הסיכוי שהעומס של אותו מינימייזר יקטן.

בחנו את היעילות של AdaOrder שלנו בכמה דרכים בהשוואה לסדרים אחרים המוכרים בספרות: סדר לכסיקוגרפי, סדר אקראי, סדר מבוסס תדירות וכן סדר חתימה, המותאם למאפייני דנא. סדר מבוסס תדירות השיג עומס מקסימלי נמוך יותר מהסדר של AdaOrder בבדיקות שביצענו, ושניהם היו טובים משמעותית במגוון קריטריונים לעומת הסדרים האחרים .

שילבנו את השיטה שלנו ב-Gerbil, התוכנה המובילה לספירת k-יות כיום, והצלחנו לצמצם את הזיכרון שלה ב-30% - 50% עבור k גדול על מגוון רצפים אמיתיים, עם עלייה קלה בלבד בזמן הריצה. ההטמעה של הסדר של AdaOrder ב- Gerbil הובילה לצריכת זיכרון נמוכה יותר מאשר ההטמעה של סדר מבוסס תדירות.

הבדיקות שלנו גם הראו כי סדרים שהופקו בשיטה שלנו על קלט מסויים הניבו תוצאות מעולות כשהם הופעלו על רצפי קלט אחרים מאותו מין (species), ללא שינוי כלשהו בסדר או עם שינוי קל ומהיר. שימוש כזה מאפשר להשיג את ההפחתה בזיכרון ללא כל עלייה בזמן הריצה.

אוניברסיטת תל אביב

הפקולטה למדעים מדוייקים ע"ש ריימונד ובברלי סאקלר

בית הספר למדעי המחשב ע"ש בלווטניק

הפחתת הזיכרון בתהליך ספירת k-יות ברצף דנא באמצעות בחירת סדר מינימייזרים המבוסס על הקלט

חיבור זה הוגש כעבודת גמר לתואר 'מוסמך אוניברסיטה'

בבית הספר למדעי המחשב על ידי

דן פלומין

בהנחיית

פרופ' רון שמיר

ניסן תשפ"ב