Tel-Aviv University

Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Improving the efficiency of de Bruijn graph construction using compact universal hitting sets

Thesis submitted in partial fulfillment of graduate requirements for

The degree "Master of Sciences" in Tel-Aviv University

School of Computer Science

By

Yael Ben-Ari

Prepared under the supervision of

Prof. Ron Shamir

Dr. Yaron Orenstein

October 2021

Acknowledgements

I would like to use this short note to express my deep gratitude to the people who accompanied me on this journey, those that I have worked with and those who supported me and helped to make it happen.

First and foremost, I would like to thank my outstanding supervisor, Prof. Ron Shamir, for supporting me in this journey. I am honored and feel fortunate to have been advised by a living legend like Ron, an exceptional researcher, a great teacher and a kind-hearted person. I felt inspired every day from his high professionalism, persistence and thoroughness.
Thank you, Ron, for believing in me and for giving me the opportunity to learn from you in the last few years.

Secondly, I would like to thank Dr. Yaron Orenstein for his great contribution to this research and for sharing with me his exceptional knowledge and experience.

Thirdly, special thanks to Dr. Lianrong Pu and Dan Flomin, who in addition to being great friends, collaborated with me in this research and helped me to make it happen. You are awesome.
I would also like to deeply thank to the rest of the lab members: Ron Z., Dvir, David, Tom, Nimrod, Hagai, Naama, Omer, Hadar, Yonatan, Eran, Roi, and Maya for being my mates during my scientific way. I am grateful for the journey we went together.
A special thanks to Gilit Zohar-Oren for her endless support that always came with a smile.

I would like to thank the agencies that supported my thesis research: the Israeli Science Foundation grant 1339/2018, and grant No. 3165/19, within the Israel Precision Medicine Partnership program, the German-Israeli Project DFG RE 4193/1-1, and Edmond J. Safra Center for Bioinformatics at Tel-Aviv University.

And last but not the least, I would like to thank my dear family: my parents Riva and Ofer and my brothers Guy and Shani. You know this would never have happened without you.

Table of Contents

Abstract

High-throughput sequencing techniques generate large volumes of DNA sequencing data at ultra-fast speed and extremely low cost. Therefore, sequencing techniques have become ubiquitous in biomedical research and are used in hundreds of genomic applications. Efficient data structures and algorithms have been developed to handle the large datasets produced by these techniques. The prevailing method to index DNA sequences in those data structures and algorithms is by k-mers (k-long substrings) known as minimizers.

Minimizers are the smallest k-mers selected in every consecutive window of a fixed length in a sequence, where the smallest is determined according to a predefined order, e.g., lexicographic. Recently, a new k-mer order based on a universal hitting set (UHS) was suggested. While several studies have shown that orders based on a small UHS have improved properties, the utility of using a small UHS in high-throughput sequencing analysis tasks has not been demonstrated to date.

Here, we demonstrate the practical benefit of UHSs for the first time, in the genome assembly task. Reconstructing a genome from billions of short reads is a fundamental task in high-throughput sequencing analyses. De Bruijn graph construction is a key step in genome assembly, which often requires very large memory and long computation time. A critical bottleneck in this process is the partitioning of DNA sequences into bins. The sequences in each bin are assembled separately, and the final de Bruijn graph is constructed by merging the bin-specific subgraphs. We incorporated a UHS-based order in the bin partition step of the Minimum Substring Partitioning algorithm of Li et al. (2013). Using a UHS-based order instead of lexicographic or random-ordered minimizers produced lower density minimizers with more balanced bin partitioning, which led to a reduction in both runtime and memory usage.

1. Introduction

Large amounts of DNA sequencing data are generated today in almost any biological or clinical study. Due to the low cost of sequencing, it has become standard to probe and measure molecular interactions and biomarkers using DNA read quantities [1]. Technologies based on high-throughput sequencing (HTS) have been developed for the major genomics tasks: genetic and structural variation detection, gene expression quantification, epigenomic signal quantification, protein binding measurements, and many more [2].

A first step in utilizing all these data types is the computational analysis of HTS data. Key challenges include read mapping to a reference genome, read compression, storing reads in a data structure for fast querying and finding read overlaps. As a result, many computational methods were developed to analyze HTS data, and the development of new methods is ongoing [3].

Many methods for analyzing HTS data use minimizers to obtain speed-up and reduce memory usage [4]– [6]. Given integers w and k, the *minimizer* of an $L = w + k - 1$-long sequence is the smallest k-mer among the w contiguous k-mers in it, where the smallest is determined based on a predefined order, e.g., lexicographic [7]. For a sequence longer than L, all L-long windows are scanned, and the minimizer is selected in each one (**Figure 1a**).

Using the minimizers to represent the L-long windows has three key advantages:

(i) The sampling interval is small.

(ii) The same k-mers are often selected from overlapping windows.

(iii) Identical windows have the same minimizer.

Minimizers help design algorithms that are more efficient in both runtime and memory usage by reducing the amount of information that is processed while losing little information. Minimizers were shown to be helpful and are used in many different settings, such as partitioning input sequences [5], [8], [9] generating sparse data structures [10], [11], and sequence classification [6].

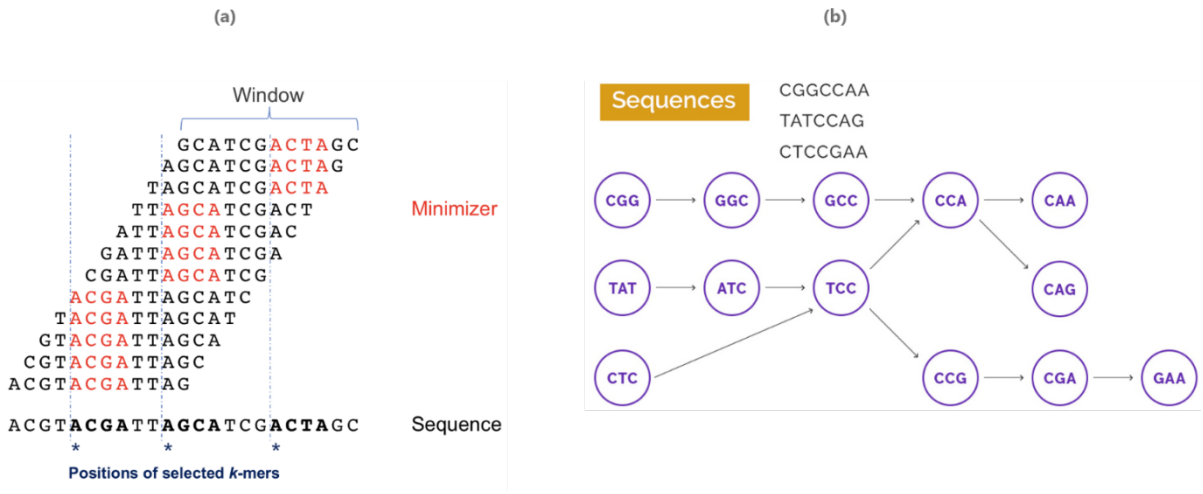

Figure 1: Illustrations of Minimizers and de Bruijn graph. (A) Minimizers scheme ($k = 4, w = 9$). The input sequence is broken into windows of length $L = w + k - 1 = 12$, and the minimizer in each window is selected. Consecutive windows tend to select the same minimizer. The positions of the selected k-mers constitute a sampling of the original sequence. (B) De Bruijn graph of order 3 for three DNA sequences. The vertices are the 3-mers contained in the set of sequences. Edges connect two vertices if the 4-mer they represent is contained in a sequence in the set.

Recently, the concept of a universal hitting set (UHS) was introduced as a way to improve minimizers [12]. For integers k and L, a set of k-mers $U_{k,L}$ is called a UHS if every possible sequence of length L contains at least one k-mer from $U_{k,L}$ as a contiguous substring. It was shown that by using a UHS of small size, one can design an order for a minimizer scheme that

results in fewer selected k-mers compared to the orders commonly used in current applications (i.e., lexicographic or random orders) [13]. Therefore, using UHSs has the potential to provide smaller signatures than currently used orders, and as a result reduce runtime and memory usage of sequencing applications.

We and others recently developed algorithms to generate small UHSs [12], [13], but so far, the prevailing methods in HTS analysis employ a lexicographic or random order. To date, only one method has been developed to take advantage of the improved properties of UHSs and applied them to a k-mer counting application [14].

In this study we demonstrate the practical benefit of UHSs on a main HTS analysis task: de Bruijn graph construction for genome assembly by a disk-based partition method. We introduce a UHS into the graph construction step of the Minimum Substring Partition assembly algorithm [15]. The introduction of the UHS into the algorithm defines a new minimizers ordering, substantially changing the execution of all the steps of the algorithm but producing exactly the same final output. In tests on several genomic datasets, the new method had lower memory usage, shorter runtime and more balanced disk partitions. The code of our method is publicly available at github.com/Shamir-Lab/MSP_UHS.

A preliminary version of this study was published in the Proceedings of the ACM-BCB Conference 2021 [16]

2. Biological Background

2 A. Genomics

DNA (Deoxyribonucleic acid) is a molecule composed of nucleotides (A -Adenine, G- Guanine, C- Cytosine, and T- Thymine) that carries all the genetics information and instructions for the development, function, reproduction and growth of all known organisms. Genes are a basic unit of heredity. They are segments along the DNA molecule that encode for the synthesis of a gene product, an RNA sequence, which can later be translated to protein. The genome is the total genetic material of an organism and includes both the genes and non-coding sequences. The Genomics field focuses on the sequence, function, evolution, mapping, and editing of genomes. Using high-performance computing and bioinformatics applications, genomics researchers analyze enormous amounts of DNA sequences to find variations that affect health, disease, drug response and more.

2 B. DNA sequencing

DNA sequencing is the process of determining the order of the nucleotides in a DNA sequence. A variety of methods and technologies are used to determine the order of the four bases. Genomics involves the sequencing and analysis of genomes through uses of DNA sequencing to assemble and analyze the function and structure of entire genomes. DNA sequencing information is used for numerous applications in molecular biology, evolutionary biology, metagenomics, medicine and many more fields.

2 C. High-throughput sequencing technologies

High-throughput sequencing technologies, also known as next-generation sequencing (NGS), are DNA sequencing techniques developed in the last fifteen years, which have completely revolutionized genome analysis. In 1977 the first sequencing method, Sanger sequencing, was developed. Around the year 2000, the whole human genome was sequenced for the first-time using Sanger sequencing. Only a few years later high-throughput sequencing techniques started to emerge and changed "the rules of the game" in Biology and Medicine. The major advantage of these techniques is the ability to sequence massively, cheaply and in parallel. These techniques create dozens to hundreds of gigabytes consisting of short DNA sequences in a single experiment, at ultrafast speeds and extremely low cost. As a result, they have become ubiquities in biomedical research. There is no doubt that high-throughput sequencing technologies accelerated the biological and biomedical research.

Since high throughput sequencing technologies generate very large amounts of data, they represent great challenges in data analyses, storage, transfer and more. These challenges have been answered by creation of hundreds of bioinformatics applications to date. These applications include read alignment, genome assembly, read mapping, single nucleotide variant detection, and many more. Ongoing improvement to the algorithms is needed as high throughput sequencing continue to develop and create more data.

2 D. Genome assembly

DNA sequencing technologies cannot read a whole chromosome in one go, but rather can read only small pieces of between 20 and 30,000 nucleotides, depending on the technology. The

location of the segments within the genome is lost in the sequencing process. Hence, sequence assembly is needed to put together and merge fragments in order to reconstruct the original genome. A main challenge in the assembly task is that it requires a huge amount of memory and very long processing time, especially for large genomes.

3. Computational Background and Definitions

Basic definitions

A *read* is a string over the DNA alphabet $\Sigma = \{A, C, G, T\}$.

A k-mer is a string of length k over Σ.

Given a read $s, |s| = n$, $s[i, j]$ denotes the substring of s from the i-th character to the j-th character, both inclusive. (Here and throughout, substrings are assumed to be contiguous.)

s contains $n - k + 1$ k-mers: $s[0, k-1]$, $s[1, k]$ … $s[n-k, n-1]$.

Two k-mers in s that overlap in $k - 1$ letters, i.e., $s[i, k + i - 1]$ and $s[i + 1, k + i]$ are called *adjacent* in s.

3 A. Minimizers schemes

Minimizers schemes are methods for selecting the smallest k-mer substrings from a sequence using predefined order, e.g., lexicographic. In an era of exponential data growth, these methods are in use in many bioinformatics due to their ability to yield a sub-linear representation of sequences, enabling sequence comparison in reduced space and time. A key property of the minimizer method is that if two sequences share a substring of a specified length, then they can be guaranteed to have a matching minimizer.

An *order* o on Σ^k is a one-to-one function $o : \Sigma^k \rightarrow \{1, 2, ..., |\Sigma|^k\}$. k-mer m_1 is smaller than k-mer m_2 according to order o if: $o(m_1) < o(m_2)$. In other words, an order is a permutation on the set of all k-mers.

A *minimizer* for a triplet (s, o, k) is the smallest k-long substring m in sequence s according to order o. We also call m the *o-minimizer* k-mer in s.

A *minimizers scheme* is a function $f_{k,w}$ that selects the start position of a minimizer k-mer in every sequence of length $L = w + k - 1$, i.e., $f: \Sigma^{w+k-1} \rightarrow [0: w-1]$ (**Figure 1a**).

3 B. De Bruijn graph

Given a set of m strings $S = \{S_0, S_1, S_2, \ldots, S_{m-1},\}$ over Σ and an integer $k \geq 2$, the *de Bruijn graph* of S of order k (**Figure 1b**) is a directed graph $dBG_k(S) = (V, E)$ where:

$$V = v \in \Sigma^k \mid \exists j \in \{0, 1, \ldots, m-1\} \; such \; that \; v \; is \; a \; substring \; of \; S_j.$$

$$E = (u, v) \mid u = S_j[i, k+i-1], v = S_j[i+1, k+i] \; for \; some \; j \; and \; i.$$

Modern genome assembly algorithms are based on de Bruijn graph construction. This process breaks each input read into k-mers (vertices in the graph) and then connects adjacent k-mers according to their overlap relations in the reads (edges). The graph represents the reconstructed genome. This process can assemble very large quantities (even billions) of reads. In genome assembly algorithms, the de Bruijn graph construction step is the most memory consuming and time-intensive part [15].

3 C. Ordering schemes

The original minimizers scheme compares k-mers using the lexicographic ordering. However, lexicographic ordering was shown to be problematic for some applications involving DNA sequences, due to over-representation of As and runs of As in the sequence. Hence, many alternatives were suggested for ordering schemes in genomic applications. One common alternative is an order determined by a random permutation of the k-mers. In Kraken application [9], for example, as a form of randomization, the authors perform bitwise XOR of the k-mers with a random value and order the resulting binary numbers lexicographically.

3 D. Particular density

We denote the set of selected positions of a scheme $f_{k,w}$ on a string S *by*

$M_{f,k,w}(S) = \{i + f_{k,w}(S[i, i + k + w - 2])$ where $0 \leq i \leq |S| - w - k + 1\}$

(These are the positions marked by asterisks in **Figure 1a**).

The *particular density* of a scheme $f_{k,w}$ on a string S is the proportion of k-mers selected:

$$d_{f,k,w}(S) = \frac{\left|M_{f,k,w}(S)\right|}{|S| - w + 1}$$

Particular density was used in previous works (e.g., [7]) as a measure of efficiency of the scheme on a particular sequence. The trivial upper and lower bounds for the density are: $\frac{1}{w} \leq d_{f,k,w} \leq 1$, where $\frac{1}{w}$ corresponds to scanning the sequence from left to right and selecting exactly one position in every new non-overlapping window, and 1 corresponds to selecting every position [17]. In general, lower density can lead to greater computational efficiency and is therefore desirable.

3 E. Universal Hitting Sets

We say that a set of k-mers M *hits* sequence S if there exists a k-mer in M that is a substring in S. A *universal hitting set* (UHS) $U_{k,L}$ is a set of k-mers that hits every L-long string over Σ. A trivial UHS always exists by taking all $|\Sigma|^k$ k-mers. A UHS M can be used in a minimizers scheme as follows:

- Define an order on M's k-mers.
- For any L-long window, select the minimum k-mer from M in the window according to the pre-defined order.

The universality of M guarantees that there will always be at least one k-mer from M in any L-long window.

3 E (i) DOCKS and PASHA algorithms

A key challenge is finding a minimum cardinality UHS, since smaller UHSs will tend to have smaller number of k-mers as minimizers on a particular sequence. It was proven that the problem of hitting a given set of L-long sequences is a NP-hard problem [18]. DOCKS and PASHA are heuristics that address the problem of finding a minimum-size UHS: They find a compact but not necessarily optimal universal k-mer set that hits any set of L-long sequences.

DOCKS

The DOCKS (Design Of Compact universal k-mer hitting Set) algorithm [12] takes as input a list of parameters (Σ, k, L) and outputs a list of k-mers, the UHS $U_{k,L}$. The algorithm has two phases:

(i) Finding a minimum-size k-mer set that hits every infinite sequence (and in particular every cycle. For that reason, the set is called a *decycling set*). This problem can be solved to optimality in polynomial time [19]

(ii) Greedily adding k-mers that hit many remaining L-long sequences until no such sequences remain. This process is heuristic and is done iteratively using dynamic programming.

The software and solution sets are freely available at acgt.cs.tau.ac.il/docks/.

PASHA

PASHA [18] is a randomized parallel algorithm for finding small UHSs. The authors build on the DOCKS algorithm and improve the calculation of the *k-mer hitting number* in a de Bruijn graph, the number of L-long strings containing the k-mer. They leveraged advanced theoretical and architectural techniques to parallelize and decrease memory usage in calculating k-mer hitting numbers. As a result, PASHA can handle larger values of k than DOCKS. The authors empirically showed that PASHA produces sets that are slightly larger than those of serial deterministic algorithms like DOCKS.

The software and solution sets are freely available at github.com/ekimb/pasha.

3 F. Relevant high-throughput sequencing applications

3 F (i). Assembly algorithms

One of the main challenges in high-throughput sequencing is assembling a massive number of short reads that were extracted from DNA segments (see the review in [18]). De novo assembly approaches particularly focus on grouping short reads into significant contigs and assembling these contigs into bins to reconstruct the original, previously unknown genomic DNA. One popular approach to assemble large genomes is by constructing a de Bruijn graph. The de Bruijn graph approach breaks short reads into k-mers and then connects k-mers according to their overlap relations in short reads. It can assemble large quantities (even billions) of short reads. Despite the broad usage of de Bruijn graphs in genome assembly applications, the large memory usage and long runtime are still a critical challenge in the de Bruijn graph construction task.

3 F (ii). Minimum Substring Partitioning (MSP)

The Minimum Substring Partitioning (MSP) method is a memory efficient and fast algorithm for de Bruijn graph construction [15]. MSP breaks reads into multiple bins so that the k-mers in each bin can be loaded into memory, processed individually to form the corresponding de Bruijn

graph, and later merged with other bins to form the full de Bruijn graph. The lexicographically smallest k-mer in each sequence window (i.e., the minimizer) is used as key for that window.

MSP partitions L-long windows into multiple disjoint bins, in a way that tends to retain adjacent L-mers in the same bin. This has two advantages:

(i) Consecutive L-mers are combined into *super L-mers* (substrings of length $\geq L$), which reduces the space requirements.

(ii) Local assembly can be performed on the bins in parallel, and later all assemblies are merged to generate a global assembly.

MSP is motivated by the fact that adjacent L-mers tend to share the same minimizer k-mer, since there is an overlap of length $L - 1$ between them. **Figure 2** shows an example of the partitioning step of MSP with $L = 10$ and $k = 3$. In this example, the first four L-mers share the minimizer AAC and the last four L-mers share the minimizer AAA. In this case, instead of generating all seven L-mers separately, MSP generates only two *super L-mers*. The first four L-mers are combined into $TGGCGAACGTAA$, and this super L-mer is assigned to the bin labeled AAC. Similarly, the last four L-mers are combined into a super L-mer $GAACCGTAAAGT$, and this super L-mer is assigned to the bin labeled AAA.

In general, given a read $r = r_0 r_1 \ldots r_{n-1}$, if the j adjacent L-mers from $r[i, i + L - 1]$ to $r[i + j - 1, i + j + L - 2]$ share the same minimizer m (and j is maximal with regard to that property), then the super L-mer $r_i r_{i+1} \ldots r_{i+j+L-2}$ is assigned to the bin labeled m without breaking it into j individual L-mers. This procedure reduces memory usage as instead of keeping $j \cdot L$ characters in memory, only $j + L - 1$ characters are kept. If j tends to be large, this strategy dramatically reduces memory usage.

To reduce the number of bins, MSP warps the bins using a hash function into a user-defined number of bins b.

Li *et al.* argued that the maximum number of distinct k-mers contained by a partition determines the peak memory. Following this reasoning, we define a bin's *load* to be the number of distinct k-mers in it, and will measure the *maximum bin load*, namely the highest load of any bin, as a criterion for peak memory usage.

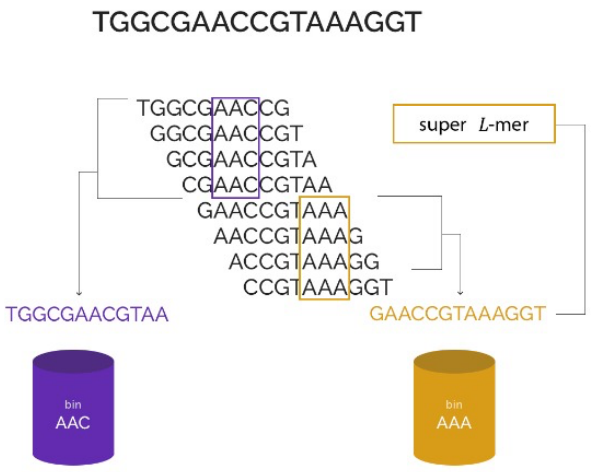

Figure 2: **The partitioning step of the MSP method**. A read is scanned in windows of length 10. The 3-mer minimizer in each window is marked with the rectangles.

4. Methods and Materials

4 A variant of MSP that uses a UHS

The original MSP algorithm uses a minimizers scheme with a lexicographic order [15]. We denote this method here *Lexico_MSP*.

Previous studies have shown that k-mers from a small UHS are more evenly distributed along the genome than lexicographic or random minimizers [12]. Hence, we reasoned that using a small UHS in the MSP algorithm would lead to a flatter distribution of bin sizes and thus reduce memory usage and runtime. We modified MSP to employ a minimizers scheme with a UHS-based order, where the order that is applied to the UHS k-mers is pseudo-random. By the definition of a UHS, a minimizers scheme based on this order selects only k-mers from $U_{k,L}$ as minimizers for any L-long window, so the order of k-mers not in $U_{k,L}$ is immaterial. We call such an order a *UHS-based minimizer order*.

We denote this algorithm *UHS_MSP*. The UHSs that we used for the algorithm were generated by DOCKS [12] for $k \leq 13$ and PASHA [21] for $k = 14$, and were taken from these algorithms' websites.

We also tested a variant of MSP where the lexicographic order is replaced by a pseudo-random order. Our reasoning was that a pseudo-random order was shown to have better properties than lexicographic order when used in a minimizers scheme [13]. We denote this variant *Random_MSP*.

UHS_MSP receives as input a set of reads and generates a corresponding de Bruijn graph by the following steps. A pseudo-code of the algorithm can be found in **Algorithm 1**.

4 A (i) Partitioning

This step uses a pre-generated UHS $U_{k,L}$. By default, we used $k = 12$ and $L = 60$. We saved $U_{k,L}$ in an array of size $|\Sigma|^k$ bits, with the values $'1'$ for the k-mers that are in $U_{k,L}$ and $'0'$ otherwise. Reads are broken into segments (super L-mers) that are placed in bins as follows: For each read, all L-long windows are scanned, and their minimizers are found. The minimizer of the currently scanned window is denoted as $currMin$ *and* its start position is denoted as $currMinPos$. The scanning is done by sliding an L-long window to the right one symbol at a time, until the end of the read. After each slide, UHS_MSP checks whether $currMinPos$ is still within the range of the current window. If not, it re-scans the window to find the current minimizer and updates $currMin$ and $currMinPos$. Otherwise, it tests whether the last k-mer in the current window is smaller than $currMin$ based on the UHS-based minimizer order. If so, the last k-mer is set as $currMin$ and its start position as $currMinPos$.

To enable fast comparison of k-mers in $U_{k,L}$, the pseudo-random order is implemented using a $2k$-long bit vector x (the *seed*), with bits selected independently and equiprobably to be $0\ or\ 1$. For $m \in U_{k,L}$ define $\beta(m) = b(m) \oplus x$, where $b(m)$ is the binary representation of k-mer m and " $\oplus$ " is the bit-wise xor operation. The order o of m is defined as the number whose binary

representation is $\beta(m)$. Hence, deciding if $o(m) < o(m')$ is done by two xor operations and one comparison.

Each time a new minimizer is selected, a super L-mer is generated by merging all the L-long windows sharing the previous minimizer, and the label of that super L-mer is its minimizer (**Figure 2**).

To obtain the prescribed number b of bins, a hash function is used to map the labels to a space of size b.

A unique ID is assigned to each L-mer when scanning the reads. As a result, identical L-mers in different positions in the data are assigned different IDs. Those will be merged in the next step.

4 A (ii) Mapping and Merging

These steps are the same as in [15]. We briefly outline them here for completeness, sine the changes we introduce in the partitioning step affect their efficiency. In the mapping step, each bin is loaded separately into the memory and identical L-mers in different positions in the bin are combined to have the same unique integer vertex ID. This process is done by generating an ID replacement table per bin. Since we expected the change in the partitioning step to create bins with sizes that are more uniformly distributed, we reasoned that the maximum bin size and the maximum memory would decrease as well.

The merging step merges the ID replacement tables of all bins and generates a global ID replacement table. The algorithm outputs sequences of IDs. Each ID is a vertex in the graph (L-mer) and two adjacent IDs represent an edge in the graph. This way, each read is represented by a sequence of the consecutive vertices of its L-mers in the graph, while identical L-mers have the same ID.

Note that the final assembled graphs of the Lexico_MSP, Random_MSP and UHS_MSP methods are identical. The differences in time and space performance are due to the particular schemes, which produce different bin size and load distributions.

Algorithm 1 UHS Minimum Substring Partitioning.

Input: A set of strings $S = (S_0, S_1, \ldots, S_{m-1})$, where $|S_i| = readLen$, integers k, L, b, a UHS-based order o for a UHS U_{kL}.

Output: The partition - b bins with the set of super L-mers in each one.

1: **for** j from 0 to $m-1$ **do**
2: $currMin$ = the o-minimum k-mer of $S_j[0, L-1]$
3: $currMinPos$ = the start position of $currMin$ in S_j
4: $currStart = 0$ /* the start position of the current super L-mer */
5: **for** i from 1 to $readLen - L$ **do**
6: **if** $i > currMinPos$ **then**
7: generate a super L-mer $sLmer = S_j[currStart, i+L-2]$
8: $currStart = i$
9: write $sLmer$ in bin number $hash(currMin)$
10: $currMin$ = the o-minimum k-mer of $S_j[i, i+L-1]$
11: $currMinPos$ = the start position of $currMin$ in S_j
12: **else**
13: **if** the last k-mer of $S_j[i, i+L-1]$ is in U_{kL} and smaller than $currMin$ **then**
14: generate a super L-mer $sLmer = S_j[currStart, i+L-2]$
15: $currStart = i$
16: write $sLmer$ in bin number $hash(currMin)$
17: $currMin$ = the last k-mer of $S_j[i, i+L-1]$
18: $currMinPos$ = the start position of $currMin$ in S_j

4 C Biological datasets

We used in our experiments four real-life datasets. Their characteristics are summarized in **Table 1.**

Dataset	Size (GB)	Avg. read length	No. of reads	Reference	Accession no.	Source
E. coli	2.9	101	20.3M	[22]	PRJNA431139	SRA
Human chr14	9.4	101	36.5M	[23]	-	GAGE
Bee	93.8	124	303M	[23]	-	GAGE
Human	432	100	2B	[24]	PRJNA29429	SRA

Table 1: Characteristics of the four benchmark datasets. All datasets were generated by the Illumina platform. The human chr14 and bee datasets were downloaded from the GAGE database (gage.cbcb.umd.edu/data/index.html).

5. Results

We compared UHS_MSP to the original MSP method (Lexico_MSP) and to MSP with random k-mer order (Random_MSP) in terms of speed, memory usage, particular density and maximum load on the four real-life datasets described in **Table 1**. The bee dataset was also used in the original MSP paper [15], but the other datasets used in that study were unavailable. We used the same parameters as in [15] for comparison, i. e., $k = 12, L = 60$, and $b = 1000$. UHS_MSP was used with the seed-based random ordering within the UHS. The use of UHS with lexicographic ordering produced inferior results and was not tested further - see **Table 3**.

Method	Lexico_MSP	Random_MSP	UHS_MSP	Lexico_UHS
Runtime (sec)	36,603	30,498	25,443	40,246
Maximum Memory(GB)	17.2	16.9	14.98	17.7

Table 3: Performance of MSP using UHS and lexicographic ordering compared to the other three algorithms on the bee dataset. The right column shows the results of MSP using UHS and lexicographic ordering.

The results correspond only to the MSP runs, and do not include the generation of the UHS, a one-time process for any values of k and L. All the experiments were measured on Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz server with 44 cores and 792 GB of RAM. To exclude the impact of parallelization, all measurements were done on a single core. On the human data set, only UHS_MSP terminated successfully and passed all the three stages

without failing. Thus, only partial results are available and reported for the other two algorithms. We note that similar problems with running MSP were reported in [19].

5 A Particular density comparison

We calculated the particular density of the MSP algorithms on the four datasets by counting the number of selected positions (unique $minPos$ in the partitioning step of the algorithm) and dividing it by the number of all possible positions. UHS_MSP achieved lower density than Lexcio_MSP and Random_MSP on all four datasets (**Table 2**). This is in accordance with the results of Marçias et al. on other genomes [13]. Lexico_MSP had the highest density. This result reaffirms the potential of UHS_MSP to achieve reduced memory usage and faster runtimes compared to the other two algorithms.

Dataset	Lexico_MSP	Random_MSP	UHS_MSP
E. Coli	0.041	0.034	0.031
Human chr14	0.073	0.065	0.062
Bee	0.059	0.056	0.053
Human	0.057	0.056	0.055

Table 2: particular density results

5 B Performance comparison

We compared the three algorithms in terms of three main performance criteria:

(1) Runtime - the total CPU time (user time + system time).

(2) Maximum memory - the maximum amount of cache memory the method used.

(3) Maximum load.

5 B (i) Runtime

Figure 3a presents the runtime of the three algorithms in seconds per GB of input data. On all datasets where comparison was possible, UHS_MSP was the fastest.

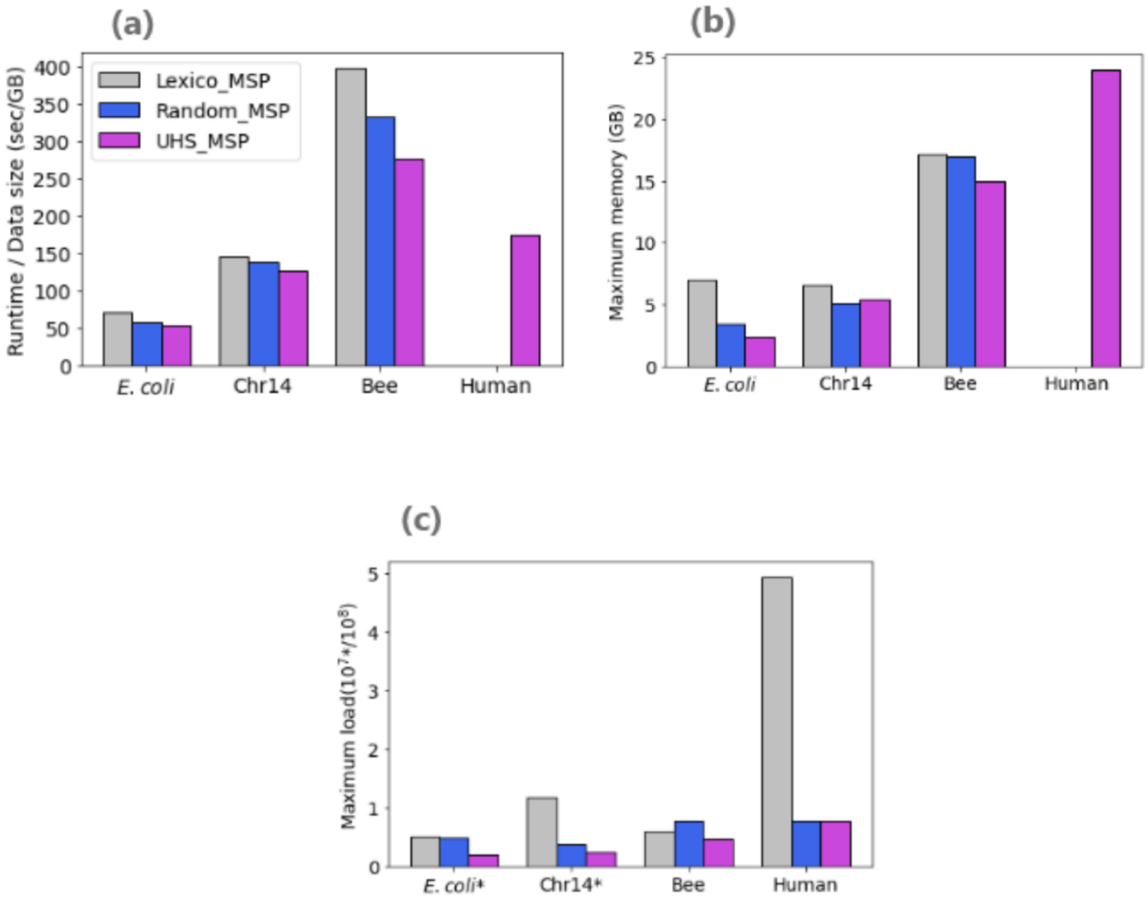

Figure 3: Tested metrics performance. (a) Runtime in seconds per GB of input data. (b) Maximum memory usage in GB. (c) Maximum load. Numbers are in 100 MB for E. coli and chr14 datasets, and in GB for the bee and human datasets. For the human dataset, the original and randomized MSP did not terminate, so runtimes and memory are not available. The reported load results are based on the partitioning and mapping steps only.

5 B (ii) Memory usage

Figure 3b displays the maximum memory used by each algorithm. UHS_MSP used substantially less memory than Lexico_MSP and achieved comparable results to Random_MSP.

5 B (iii) Maximum load

Figure 3c display the maximum load value. UHS_MSP also had the lowest maximum load on all datasets.

The results show that UHS_MSP achieved a substantial improvement over the other algorithms in all three main aspects. We also measured two additional criteria: the largest bin size and the total disk space. The results for these criteria are presented in **Figure 6**. They present consistent advantage to UHS_MSP over both Random_MSP and Lexico_MSP.

5 B iv. Test of robustness with different random orders

As an additional test of the robustness of UHS_MSP, we wished to gauge the effect of the pseudo-random order of the k-mers on the results. We ran Random_MSP and UHS_MSP, which use randomized orders, on the four datasets with five different seeds, corresponding to different pseudo-random orders (Section 3). The results are presented in **Table 3**. While both algorithms showed substantial performance variance across orders, overall, the results were in line with those presented in **Figure 3**.

	Runtime (sec)			Maximum memory (GB)			Max load (10^6)		
Data	Lexico	Random	UHS	Lexico	Random	UHS	Lexico	Random	UHS
E. coli	207	206 ±35.6	**148** ±5.54	6.96	6 ±1.3	**4** ±0.5	5.04	4.05±3.8	**1.3**±0.7
Human chr14	1371	1349±38.2	**1189** ±68.13	6.66	5.55±0.83	**5.12**±0.73	11.8	**2.3**±0.9	2.36±0.07
Bee	36603	28159±12815	**24114**±10836	17.2	15.94±1.02	**15.63**±0.44	58.4	51.3±19	**37.9**±9.1
Human	NA	NA	**77528** ±3536	NA	NA	**28** ±0.9	431	**69.3** ±3	74.9 ±2.3

Table 3: Performance across different pseudo-random *k*-mer orders. Average and standard deviation over five runs with different seeds are shown for the two algorithms that use a random order, alongside the original MSP results. On the human dataset, the original MSP as well as its pseudo-random order version did not terminate successfully.

5 C The effect of parameters k, L and b

We tested the three methods on the bee data in a range of values for the parameters k, L and b. In each run, we kept two of the three parameters at their default values and varied the third. The results are summarized in **Figure 4**. Changing the number of bins shows consistent advantage to UHS_MSP, with a tendency to improve as the number of bins increases **(a-c).** Changing L shows a similar advantage to UHS_MSP **(d-f)**. Changing k has a less consistent effect **(g-i)**. **Figure 7** shows the effect of changing each of the three parameters on the largest bin size. Again, UHS_MSP was consistently better across the tested values of L and b, and less so when changing k.

Li et al. also tested on Lexico_MSP the impact of changing k and L [20]. The impact of varying k was consistent with what we observed here for that algorithm **(Figure 4g-i)** with reduction of resources needed as k increases. They also reported a similar reduction when L increases, unlike a less consistent picture observed here **(d-f)**. Note however that they tested the range $L =$

$31 - 63$ while we tested a broader range of higher values $L = 60 - 120$. While varying the number of bins was not tested before, our tests here **(a-c)** show improved performance with increasing b, and a similar trend (with leveling off at high values) by the two other algorithms as well.

Previous studies also tested for the impact of the minimizer's length k in similar minimum-substring-partitioning applications, and recommended values such as [21], [22] and [23]. Our experiments also tested for k values twice as large.

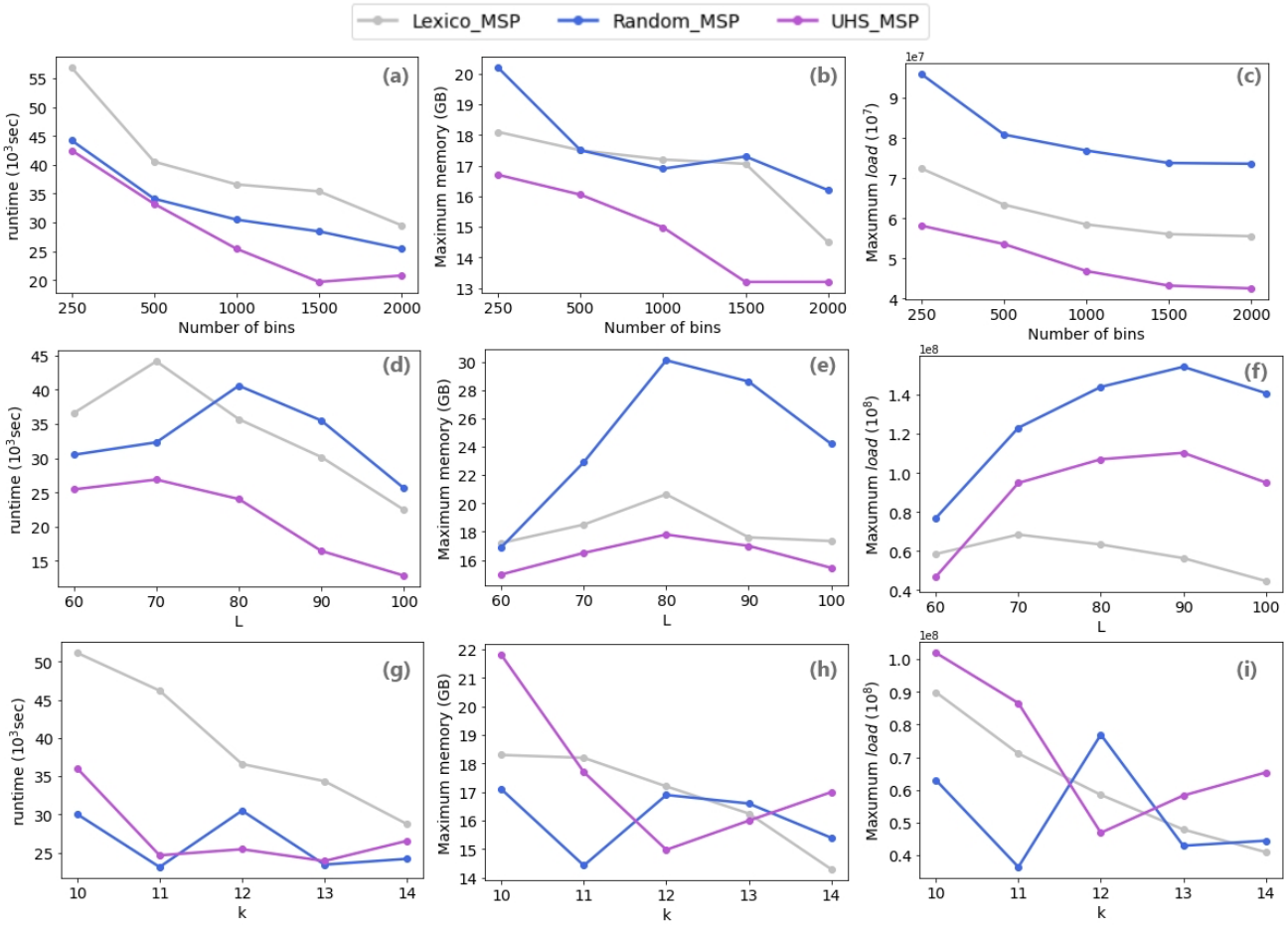

Figure 4: The effect of changing the number of bins, the window size and the k-mer size on performance. Results are for the bee dataset. (a-c) Effect of the number of bins. (d-f) Effect of the window size L. (g-i) Effect of k. For each parameter, the runtime, maximum memory and the maximum load are shown.

5 D Resource usage in each step of the algorithm

To appreciate where the saving is achieved, we measured the resources consumed by each of the three MSP steps: partitioning, mapping and merging. **Figure 5** summarizes the results on the

bee dataset. The mapping step required most memory, taking an order of magnitude more memory than the partitioning and 3-5-fold more than the merging step. In all three algorithms, the mapping step was also the most time-demanding one, taking on average 67% of the time. The merging step required 20% of the time, on average, and the partitioning step was the least demanding one, taking 13% of the time. Remarkably, even though we explicitly changed only the partitioning step of the algorithm, that change led to substantial reduction in the time and memory of the mapping step. The merging step was less affected. In comparison to the original MSP algorithm, UHS_MSP required 4% more time in the partitioning step, due to the extra work required for UHS-related computations but was 20% faster in the mapping step. Note that for the sake of our tests, we did not utilize the possibility of parallelizing the mapping step. Future work can thus focus on improvements to the mapping step.

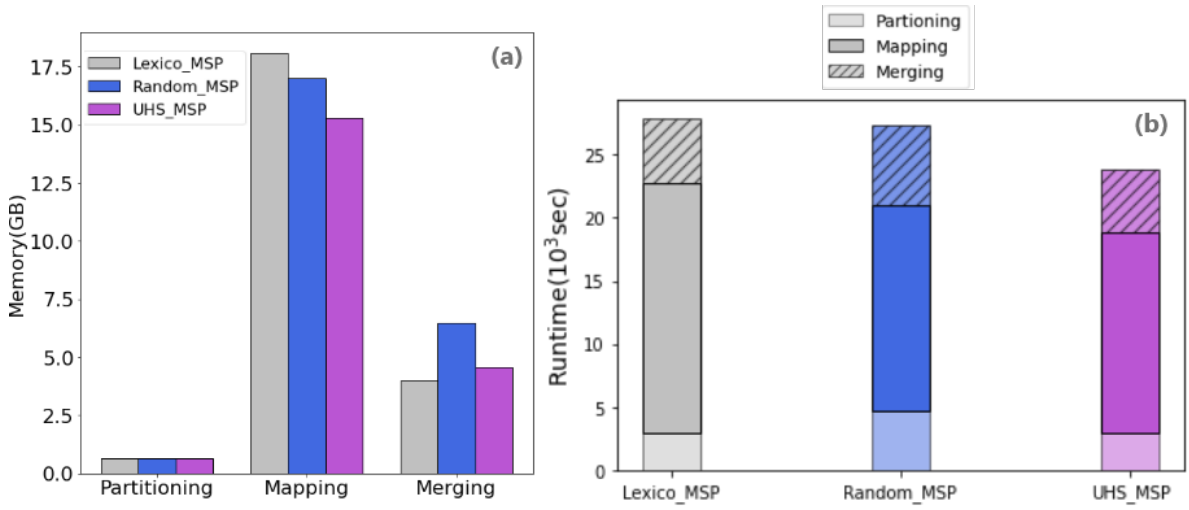

Figure 5: Resources taken by each algorithm and each step of the algorithm on the bee dataset. (a) Maximum Memory. (b) Runtime.

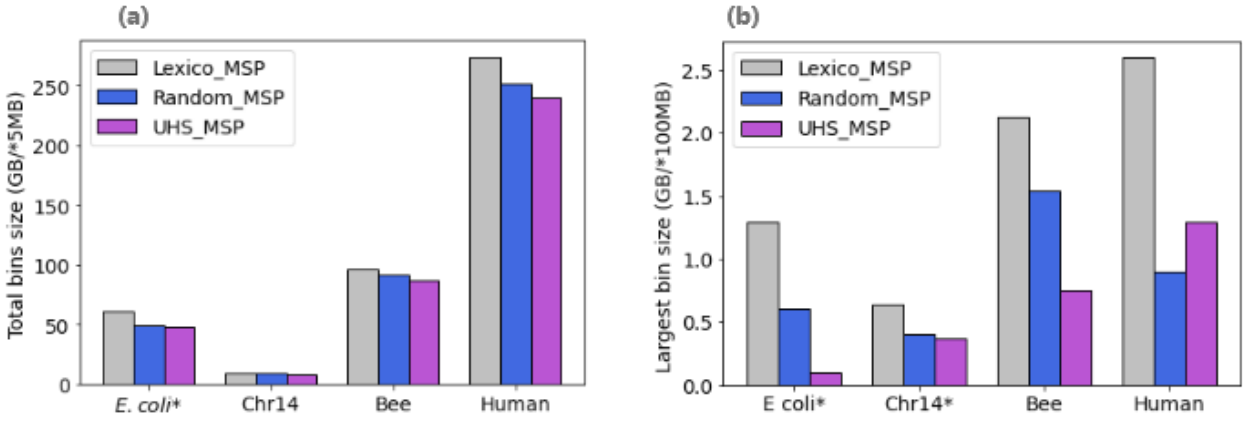

Figure 6: Additional tested metrics performance. (a) Total bins size. The total size of the bins stored on the disk is shown. Numbers are in 5MB for *E.coli* dataset and in GB for chr14, bee and human datasets. (b) Largest bin size. Numbers are in 100 MB for *E. coli* and chr14 datasets, and in GB for the bee and human datasets.

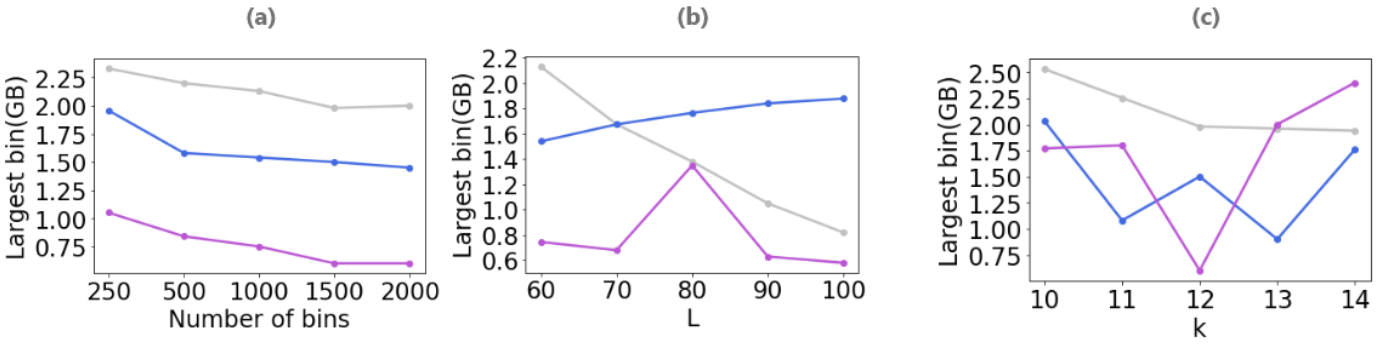

Figure 7: The effect of changing the number of bins, the window size and the k-mer size on the largest bin size. Results are for the bee dataset. (a) Effect of the number of bins. (b) Effect of the window size L. (c) Effect of k.

6. Discussion

In this study, we incorporated a UHS-based minimizers scheme in a fundamental HTS task: de Bruijn graph construction. By creating partitions based on fewer k-mers and with better statistical properties, we achieved speedups and reduced memory usage in genomic assembly. To the best of our knowledge, this is the first demonstration of the practical advantage of using UHSs in a genome assembly application.

Our study raises several open questions: to what extent can further improvements in the generation of smaller UHSs improve the de Bruijn graph construction? Currently the complexity of minimum size UHS still remains an open problem, though closely related problems were shown to be computationally hard [12], [13]. Can one express the expected amount of resources needed by the UHS_MSP algorithm (and by its separate steps), as a function of the key parameters k, L and b? Obtaining such an estimate, even under a simple model such as the random string mode, can guide one to optimize the combination of parameter values, which as we have seen tend to interact in a rather complex way (**Figure 4**). What is the relation between the largest bin size and the maximum memory usage? Can one improve the mapping of multiple minimizers into a bin? Some applications (e.g., [8]) sample the input data in order to map minimizers to bins more efficiently. Integrating such a method into MSP can improve memory and runtime performance. Li et al. [15] argued that the maximum load determines the peak RAM consumption. Our results were not consistent with this claim. (For example, see the results on the bee data in **Figure 3** and **Figure 4 (d-f)**). Further investigation revealed that some of the incoherence is due to Java's garbage collector. Changing Java's limit for the heap size reduced memory consumption substantially. For example, when running the chr14 dataset with a limit of 4GB, the peak memory

went down from 6.6 GB to 2.7 GB, with no increase in running time. In other runs even larger peak memory changes were observed depending on the limit, but at the expense of longer runtimes. Since the measured memory of a Java process is not reliable, and depends on the machine and on Java's memory flags, an implementation in a language with explicit memory management (e.g. c++) is preferable.

In parallel to our study, a work by Nystrom-Persson et al. [14] also implemented UHS in a sequencing application. The authors combined UHS and frequency counts in a k-mer counting application and achieved substantial memory saving. The improvement now achieved by UHSs in two different sequencing applications is encouraging. It is tempting to believe that practical improvements can be achieved in other applications that utilize minimizers, e.g. $BCALM2$ [24] (which utilizes frequency-based minimizers, given the results in [25]), applications that perform partitioning of sequences as a preprocessing step for efficient parallel processing and storage [21], [24], [26], sequence similarity estimation [27], [28], and others.

7. References

[1] J. A. Reuter, D. v. Spacek, and M. P. Snyder, "High-Throughput Sequencing Technologies," *Molecular Cell*, vol. 58, no. 4. Cell Press, pp. 586–597, May 21, 2015. doi: 10.1016/j.molcel.2015.05.004.

[2] W. Huber *et al.*, "Orchestrating high-throughput genomic analysis with Bioconductor," *Nature Methods*, vol. 12, no. 2, 2015, doi: 10.1038/nmeth.3252.

[3] S. Anders, P. T. Pyl, and W. Huber, "HTSeq-A Python framework to work with high-throughput sequencing data," *Bioinformatics*, vol. 31, no. 2, 2015, doi: 10.1093/bioinformatics/btu638.

[4] G. Kucherov, "Evolution of biosequence search algorithms: a brief survey," *Bioinformatics*, 2019.

[5] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[6] D. E. Wood and S. L. Salzberg, "Kraken: ultrafast metagenomic sequence classification using exact alignments," *Genome Biology*, vol. 15, no. 3, p. R46, 2014.

[7] 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*, 2003, pp. 76–85.

[8] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[9] M. Roberts, B. R. Hunt, J. A. Yorke, R. A. Bolanos, and A. L. Delcher, "A preprocessor for shotgun assembly of large genomes," *Journal of Computational Biology*, vol. 11, no. 4, pp. 734–752, 2004.

[10] S. Grabowski and M. Raniszewski, "Sampling the suffix array with minimizers," in *International Symposium on String Processing and Information Retrieval*, 2015, pp. 287–298.

[11] C. Ye, Z. S. Ma, C. H. Cannon, M. Pop, and W. Y. Douglas, "Exploiting sparseness in de novo genome assembly," in *BMC Bioinformatics*, 2012, vol. 13, no. 6, p. S1.

[12] 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*, vol. 13, no. 10, p. e1005777, 2017.

[13] G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford, "Improving the performance of minimizers and winnowing schemes," *Bioinformatics*, vol. 33, no. 14, pp. i110–i117, 2017.

[14] J. Nyström-Persson, G. Keeble-Gagnère, and N. Zawad, "Compact and evenly distributed k-mer binning for genomic sequences," *Bioinformatics*, Sep. 2021, doi: 10.1093/bioinformatics/btab156.

[15] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, "Memory efficient minimum substring partitioning," in *Proceedings of the VLDB Endowment*, 2013, vol. 6, no. 3, pp. 169–180.

[16] 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*, 2021, pp. 1–9.

[17] Y. Li and others, "MSPKmerCounter: a fast and memory efficient approach for k-mer counting," *arXiv preprint arXiv:1505.06550*, 2015.

[18] Y. Orenstein, D. Pellow, G. Marçais, R. Shamir, and C. Kingsford, "Compact universal k-mer hitting sets," in *International Workshop on Algorithms in Bioinformatics*, 2016, pp. 257–268.

[19] J. Mykkeltveit, "A proof of Golomb's conjecture for the de Bruijn graph," *Journal of Combinatorial Theory, Series B*, vol. 13, no. 1, pp. 40–45, 1972, doi: https://doi.org/10.1016/0095-8956(72)90006-8.

[20] J. Sohn and J.-W. Nam, "The present and future of de novo whole-genome assembly," *Briefings in bioinformatics*, vol. 19, no. 1, pp. 23–40, 2018.

[21] B. Ekim, B. Berger, and Y. Orenstein, “A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets,” *bioRxiv*, 2020, doi: 10.1101/2020.01.17.910513.

[22] N. I. Johns *et al.*, “Metagenomic mining of regulatory elements enables programmable species-selective gene expression,” *Nature methods*, vol. 15, no. 5, pp. 323–329, 2018.

[23] S. L. Salzberg *et al.*, “GAGE: A critical evaluation of genome assemblies and assembly algorithms,” *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.

[24] D. R. Bentley *et al.*, “Accurate whole human genome sequencing using reversible terminator chemistry,” *nature*, vol. 456, no. 7218, pp. 53–59, 2008.

[25] S. C. Manekar and S. R. Sathe, “A benchmark study of k-mer counting methods for high-throughput sequencing,” *GigaScience*, vol. 7, no. 12, Sep. 2018, doi: 10.1093/gigascience/giy125.

[26] J. A. Reuter, D. v Spacek, and M. P. Snyder, “High-throughput sequencing technologies,” *Molecular Cell*, vol. 58, no. 4, pp. 586–597, 2015.

[27] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, “KMC 2: Fast and resource-frugal k-mer counting,” *Bioinformatics*, vol. 31, no. 10, 2015, doi: 10.1093/bioinformatics/btv022.

[28] M. Erbert, S. Rechner, and M. Müller-Hannemann, “Gerbil: A Fast and Memory-Efficient k-mer Counter with GPU-Support,” *CoRR*, vol. abs/1607.06618, 2016, [Online]. Available: http://arxiv.org/abs/1607.06618

[29] M. Kokot, M. Długosz, and S. Deorowicz, “KMC 3: counting and manipulating k-mer statistics,” *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, Sep. 2017, doi: 10.1093/bioinformatics/btx304.

[30] R. Chikhi, A. Limasset, and P. Medvedev, “Compacting de Bruijn graphs from sequencing data quickly and in low memory,” *Bioinformatics*, vol. 32, no. 12, 2016, doi: 10.1093/bioinformatics/btw279.

[31] J. Nyström-Persson, G. Keeble-Gagnère, and N. Zawad, “Compact and evenly distributed k-mer binning for genomic sequences,” *bioRxiv*. bioRxiv, p. 2020.10.12.335364, Oct. 12, 2020. doi: 10.1101/2020.10.12.335364.

[32] Y. Li and X. Yan, “MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting,” 2015. Accessed: Mar. 25, 2021. [Online]. Available: http://www.cs.ucsb.edu/~yangli/MSPKmerCounter

[33] C. Jain, A. Dilthey, S. Koren, S. Aluru, and A. M. Phillippy, “A fast approximate algorithm for mapping long reads to large reference databases,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10229 LNCS. doi: 10.1007/978-3-319-56970-3_5.

[34] B. D. Ondov *et al.*, “Mash: Fast genome and metagenome distance estimation using MinHash,” *Genome Biology*, vol. 17, no. 1, p. 132, Jun. 2016, doi: 10.1186/s13059-016-0997-x.

לקסיקוגרפי או סידור אקראי יצר מינימייזרים בצפיפות נמוכה יותר ועם חלוקה מאוזנת יותר לקבוצות. תכונות משופרות אלו הובילו לשיפור הן בזמן הריצה והן בצריכת הזיכרון בתהליך הרכבת הגנום.

תקציר בעברית

שיטות ריצוף דנא מהדור החדש מייצרות כמויות אדירות של מידע גנומי המרוצף במהירות אדירה ובמחיר נמוך מאוד. בזכות כך, שיטות אלו נמצאות בשימוש נרחב במחקרים ביו-רפואיים. ישנם כיום מאות יישומים המשתמשים במידע גנומי שנוצר בעזרת שיטות ריצוף מהדור החדש. כדי להתמודד עם הכמות העצומה של מאגרי המידע שנוצרו בשיטות ריצוף אלה, פותחו אלגוריתמים ומבני נתונים יעילים המתמודדים עם אתגרי האחסון המידע, ניתוח המידע, העברתו ועוד.

הדרך הנפוצה ביותר למפתח רצפי דנא במבני הנתונים ובאלגוריתמים הללו היא בעזרת מחרוזות באורך k ("ק-מרים") הנקראות "מינימייזרים". המינימייזר של חלון בטקסט הוא הק-מר הקטן ביותר בו. לצורך המפתוח נבחר מינימייזר בכל חלון עוקב בגדול קבוע ברצף. ערך הק-מרים נקבע על ידי סידור שנקבע מראש (למשל, סידור לקסיקוגרפי).

לאחרונה הוצע סידור חדש של ק-מרים המבוסס על המושג של קבוצת כיסוי אוניברסלית (Universal Hitting Set). למרות שלא מעט מחקרים הראו כי לסידור מהסוג הזה יש מאפיינים משופרים, התועלת של סידור זה ביישומים המנתחים מידע גנומי רב לא הוכחה עד היום. אנחנו מציגים כאן לראשונה את היתרון והשימושיות של סידור המבוסס על קבוצת כיסוי אוניברסלית באלגוריתם להרכבת הגנום.

הרכבת הגנום ממיליארדי מקטעים קצרים היא משימה בסיסית ומרכזית באנליזת המידע הנוצר בעזרת שיטות הריצוף מהדור החדש. בניית גרף דה ברויין הינה שלב מרכזי בהרכבת הגנום, שלרוב דורש כמות גדולה מאוד של זיכרון יחד עם זמן חישוב רב. בגלל שמדובר בכמות מידע עצומה, אין אפשרות שכל המידע יימצא בו-זמנית בזיכרון, ולכן מתבצעת חלוקה של רצפי הדנ"א לקבוצות שונות כדי שכל קבוצה תעלה לזיכרון בנפרד. הרצפים בכל קבוצה מורכבים לתת גרף נפרד וגרף הדה ברויין הסופי נבנה על ידי מיזוג תתי הגרפים מכל הקבוצות. אלגוריתם מוביל לבניית גרף דה ברויין בתהליך כזה הוא MSP [13]. בעבודתנו, שילבנו סידור מבוסס קבוצת כיסוי אוניברסלית בתהליך חלוקת הרצפים לקבוצות באלגוריתם MSP. השימוש בסידור מסוג זה במקום סידור

אוניברסיטת תל-אביב

הפקולטה למדעים מדויקים ע"ש ריימונד ובברלי סאקלר

בית הספר למדעי המחשב ע"ש בלווטניק

שיפור היעילות של בניית גרף דה ברויין

בעזרת קבוצת כיסוי אוניברסלית

חיבור זה הוגש כעבודת גמר לתואר 'מוסמך׳ אוניברסיטה

בבית הספר למדעי המחשב על ידי

יעל בן-ארי

נעשה תחת ההנחיה של

פרופסור רון שמיר

ד״ר ירון אורנשטיין

אוקטובר 21