Université de Lille
École Graduée MADIS

Thèse

pour obtenir le grade de :

Docteur de l'Université de Lille

dans la spécialité

« Informatique »

par

Igor Martayan

Algorithm design and implementation for the scale of sequencing data

Conception et implémentation d'algorithmes à l'échelle des données de séquençage

Thèse soutenue le 4 septembre 2026 devant le jury composé de :

M.	Sven Rahmann	Professor, Saarland University	(Rapporteur)
M.	Alexandru Tomescu	Professor, University of Helsinki	(Rapporteur)
M.	Giulio Ermanno Pibiri	Associate Professor, Ca' Foscari University	(Examinateur)
M.	Éric Rivals	Directeur de recherche, Université de Montpellier	(Examinateur)
M.	Sylvain Salvati	Professeur, Université de Lille	(Président du jury)
Mme	Camille Marchet	Chargée de recherche, Habilitée à diriger des recherches, Université de Lille	(Directrice de thèse)

Laboratoire CRIStAL
Bâtiment ESPRIT
Avenue Henri Poincaré
59655 Villeneuve d'Ascq
France

Abstract

The management and analysis of large collections of DNA sequences represents a growing challenge for bioinformatics. Advances in sequencing technologies continue to increase both the volume and diversity of available data: public repositories now hold several petabytes of sequences, yet computational limitations in storage and indexing make this data difficult to exploit fully. How, then, should we design algorithms suited to this scale?

This thesis addresses the design and implementation of efficient algorithms for genomic sequence analysis, with an emphasis not only on theoretical efficiency but, crucially, on practical performance. We argue that algorithm design and implementation are inseparable: achieving high throughput in practice requires both to be considered together, with a deep understanding of how modern hardware executes code.

The first part focuses on high-performance sequence processing. We show that SIMD vectorization, the ability of modern CPUs to operate on multiple values simultaneously, can dramatically accelerate the core steps of a genomic processing pipeline. We present vectorized methods for sequence parsing, rolling hash computation, and minimizer extraction, and demonstrate their practical application. Together, these form a pipeline that processes genomic data at close to hardware-limited throughput.

The second part examines how the choice of data representation for substrings of fixed length, or k-mers, affects the efficiency of data structures and plays an important role in practical performances. We study the relationship between minimizers and necklaces, and show how these representations naturally group related k-mers together, enabling cache-friendly data structures. Building on this, we propose different compact representations that support efficient k-mer counting and set operations on large collections of sequences.

The third part considers sparser representations of sequence content. We show that by retaining only a carefully chosen fraction of k-mers, one can reduce both memory footprint and comparison time without sacrificing the ability to answer similarity queries.

Across these three axes, our work shows that significant practical gains come from treating algorithm design and implementation as a single, unified problem rather than two separate concerns.

Résumé en français

La gestion et l'analyse de grandes collections de séquences d'ADN représentent un défi croissant pour la bioinformatique. Les avancées des technologies de séquençage continuent d'accroître le volume et la diversité des données disponibles : les dépôts publics contiennent désormais plusieurs pétaoctets de séquences, mais des limitations computationnelles en matière de stockage et d'indexation rendent ces données difficiles à exploiter pleinement. Comment, dès lors, concevoir des algorithmes adaptés à cette échelle ?

Cette thèse s'intéresse à la conception et à l'implémentation d'algorithmes efficaces pour l'analyse de séquences génomiques, en mettant l'accent non seulement sur l'efficacité théorique, mais aussi et surtout sur les performances en pratique. Nous défendons l'idée que conception algorithmique et implémentation sont indissociables : atteindre un débit élevé en pratique exige de considérer les deux conjointement, avec une compréhension approfondie de la façon dont le hardware moderne exécute le code.

La première partie porte sur le traitement haute performance de séquences. Nous montrons que la vectorisation SIMD, c'est-à-dire la capacité des processeurs modernes à opérer simultanément sur plusieurs valeurs, permet d'accélérer considérablement les étapes clés d'un pipeline de traitement génomique. Nous présentons des méthodes vectorisées pour l'analyse syntaxique de séquences, le calcul de hash glissant et l'extraction de minimiseurs, et en démontrons l'application pratique. Ensemble, ces méthodes forment un pipeline traitant les données génomiques à un débit proche de la limite matérielle.

La deuxième partie examine comment le choix de la représentation des sous-chaînes de longueur fixe, ou k-mers, influe sur l'efficacité des structures de données et joue un rôle important dans les performances pratiques. Nous étudions la relation entre minimiseurs et colliers, et montrons comment ces représentations regroupent naturellement les k-mers similaires, permettant des structures de données favorables au cache. Sur cette base, nous proposons différentes représentations compactes permettant le comptage efficace de k-mers et les opérations ensemblistes sur de grandes collections de séquences.

La troisième partie s'intéresse à des représentations plus parcimonieuses du contenu des séquences. Nous montrons qu'en ne conservant qu'une fraction soigneusement choisie des k-mers, il est possible de réduire à la fois l'empreinte mémoire et le temps de comparaison, sans sacrifier la capacité à répondre à des requêtes de similarité.

À travers ces trois axes, nos travaux montrent que des gains pratiques significatifs découlent du fait de traiter la conception algorithmique et l'implémentation comme un problème unique et unifié, plutôt que comme deux préoccupations séparées.

Foreword

I designed this manuscript with a simple goal in mind: creating a resource I would have enjoyed reading at the beginning of my PhD. Looking for such a resource, I found that many articles focused on either algorithms or implementation, but rarely both: theoretical papers rarely connected their results to implementation choices, while engineering-focused work often lacked a clear account of the underlying ideas. This document is my attempt to combine both perspectives, building from shared foundations before focusing on contributions that make the link between design choices and practical performance explicit.

The three parts can be read independently, but together they build toward a common argument: that careful implementation matters as much as algorithm design, that memory representation plays a key role in practical performance, and that at the largest scales, sparser representations become necessary. The first part asks how fast a genomic pipeline can run when the CPU is fully put to use, and builds vectorized methods for parsing, hashing, and minimizer extraction that approach the hardware limit. The second asks how the internal representation of k-mer sets influences what is actually fast in practice, and shows that structures exploiting the locality of consecutive k-mers can improve upon generic solutions while supporting richer operations. The third asks how much information can be discarded while still answering similarity queries reliably, and shows that a carefully chosen subset of k-mers is often sufficient to preserve the answers we care about. Each part assumes only the introduction as a prerequisite, begins with additional background before focusing on new contributions, and closes with my perspectives on this work and what could come next.

An online version of this thesis is available at phd.martayan.org. I hope you'll enjoy reading it.

Acknowledgements • Remerciements

I would like to start by thanking all my international friends and colleagues I've had the chance to meet and work with, all these encounters have been an incredibly motivating factor during my PhD! Thank you Rob and everyone at UMD for giving me such a warm welcome despite the difficult times, I hope we'll have more opportunities to meet in the future. Thank you Ragnar, I'm so glad we got to know each other. Working with you has taught me so much, and our discussions have sparked countless ideas across my projects. I can't wait to meet again under the Mediterranean sun!

I also wish to express my gratitude to Sven Rahmann, Alexandru Tomescu, Giulio Ermanno Pibiri, Éric Rivals and Sylvain Salvati for kindly agreeing to review this work. The rest of these acknowledgements will be in French, because it would be a shame not to properly thank those who accompanied me along the way.

Avant tout, merci Camille de m'avoir guidé tout au long de ce voyage, depuis Rennes il y a 5 ans jusqu'à aujourd'hui. De toute évidence, j'ai eu énormément de chance de te rencontrer et je n'en serais pas là aujourd'hui sans tout ce que tu m'as apporté. Merci pour la confiance que tu m'as accordée dès le départ dans tous mes projets, et pour ton soutien dans les moments où j'en avais le plus besoin.

Merci Antoine de m'avoir accueilli à Lille pour mon stage, d'avoir élargi mes horizons culinaires et de m'avoir lancé sur une quantité innombrable de nouveaux problèmes. Je penserai à toi quand j'aurai mon autographe de Faker ! Merci Jean-Stéphane d'avoir dirigé ma thèse pendant quelques temps et de m'avoir motivé à faire de l'enseignement, ça en valait la peine !

Plus généralement, merci à toute l'équipe Bonsai (avant ou après expansion à l'intégralité du 3e étage) pour toutes ces années passées ensemble ! Merci Lilliang, Thomas et Léa de m'avoir montré qu'on pouvait survivre à la rédaction d'une thèse ! Merci Florian de m'avoir fait comprendre qu'il n'y a pas besoin de soleil tant qu'on a un lion. Merci beaucoup Tim, d'avoir été si accueillant dès que je suis arrivé, c'est en grande partie grâce à toi si je me suis senti aussi bien aussi vite. Bon courage Tim et Karl pour la dernière ligne droite, je vous soutiens sur tous les points ! Merci Lucas, mon ~~stagiaire~~ voisin préféré, de m'avoir accompagné pendant toutes ces ~~vacances~~ conférences ! Vivement la prochaine ! Merci aussi à Klára et Lore, qui doivent se demander ce que leur nom fait au milieu de ce paragraphe (well job if you've read this far). Merci à Pierre de m'avoir fait découvrir la merveilleuse ville de Besancon ! Merci à Jules, qui aura sans le savoir été l'un des tous premiers lecteurs de cette thèse (tu es capable d'accomplir de grandes choses ♫) ! Et merci à Bastien, Bastien, Mathilde, Mathilde, Yoann, Yohan, Valendin, Madeleine, Caleb, Karim, Loïc, Étienne, Simon, Alix et Louise pour toutes ces parties au fil des journées ! Merci Ilévia, d'avoir apporté une touche d'imprévu à des trajets qui auraient été autrement bien monotones !

Merci à l'École Normale Supérieure de Rennes de m'avoir donné goût à la recherche en informatique et d'avoir financé cette thèse.

Enfin merci à ma famille, particulièrement mes parents et ma sœur, d'avoir toujours cru en moi et de m'avoir soutenu toutes ces années.

Table of contents

Abstract 3

Résumé en français 4

Foreword 5

Acknowledgements • Remerciements 6

I. Introduction 11

1. Delving into genomic sequences 13

2. Comparing using *k*-mers 15
2.1. Coarse-grained comparisons with *k*-mers . . . 15
2.2. De Bruijn graph representation . . . 15
2.3. Using *k*-mers for petabase-scale data structures . . . 17

3. Sketching sequences 19
3.1. Why do we want to sketch sequences? . . . 19
3.2. Approximating similarity . . . 19
3.3. MinHash and its variations . . . 19
3.4. Partitioned sketches and smaller fingerprints . . . 21
3.5. Other types of sketches . . . 22
3.6. Toward locality-aware sampling . . . 23

4. Sampling with minimizers 25
4.1. Definition and fundamental properties . . . 25
4.2. Density and orderings . . . 25
4.3. Super-*k*-mers . . . 26
4.4. Improving conservation . . . 27
4.5. Practical use cases . . . 28

II. High-performance sequence processing 31

5. A primer on vectorization 33
5.1. A brief history of vectorized extensions . . . 33
5.2. SIMD vectorization in practice . . . 34
5.3. The example of memchr . . . 35

6. Vectorized sequence parsing 37
6.1. Background . . . 37
6.2. (Bitpacked) DNA representations . . . 38

6.3. Streaming the input . . . 39
6.4. Classifying the input with bitmasks . . . 39
6.5. Implementation of the lexing phase . . . 41
6.6. Parsing relevant information with a control finite state machine . . . 42
6.7. Compile-time configuration and specialization . . . 43
6.8. Results . . . 44
6.9. Conclusion . . . 47

7. Rolling hashes on sequences **49**
7.1. Rolling hashes and their properties . . . 49
7.2. NtHash and its variants . . . 50
7.3. Vectorized implementation of ntHash . . . 51
7.4. Breaking ntHash . . . 53
7.5. Toward minimizer computation . . . 55

8. Vectorized computation of minimizers **57**
8.1. Existing algorithms for sliding window minimum . . . 57
8.2. A branchless linear algorithm using two stacks . . . 59
8.3. Computing canonical minimizers . . . 61
8.4. Vectorized implementation . . . 62
8.5. Variations of minimizers . . . 63
8.6. Experimental evaluation . . . 64

9. Application to sequence filtering **69**
9.1. Filtering sequences with minimizers . . . 70
9.2. Results . . . 72
9.3. Limitations and perspectives . . . 77

First discussion **79**

III. Locality-preserving representations of k-mer sets **81**

10. Background on *k*-mer sets **83**
10.1. Path locality and hashing . . . 83
10.2. Colexicographic clustering and spectral BWT . . . 85
10.3. Masked superstrings . . . 87
10.4. Comparison . . . 88

11. Necklaces and minimizers **89**
11.1. Necklaces and their properties . . . 89
11.2. Necklaces of consecutive *k*-mers . . . 90
11.3. CBL: a dynamic data structure for *k*-mer sets . . . 92
11.4. Comparison against other indexes . . . 94
11.5. Perspectives . . . 96

12. Set operations on *k*-mer collections **99**
12.1. The sorted-iterator interface . . . 99
12.2. Multi-way set operations . . . 99

12.3. Prefix-level pruning . . . 100
12.4. Benchmarks . . . 100
12.5. Exponential search for skewed set sizes . . . 102
12.6. Perspectives from database query engines . . . 102

13. Dynamic super-*k*-mer maps **103**
13.1. Interleaved super-*k*-mers . . . 103
13.2. Load balancing and superbuckets . . . 106
13.3. Super-*k*-mer versus *k*-mer level representation . . . 107
13.4. Performance in practice . . . 107
13.5. Toward streaming deduplication . . . 109

14. From super-*k*-mers to hyper-*k*-mers **111**
14.1. Space analysis of super-*k*-mers and closed syncmers . . . 111
14.2. Hyper-*k*-mers . . . 113
14.3. Counting *k*-mers using hyper-*k*-mers . . . 114
14.4. Experiments . . . 115
14.5. Conclusion . . . 116

Second discussion **123**

IV. Sampling k-mers to lower memory & complexity **125**

15. Background on low-density minimizers **127**
15.1. Universal hitting sets and decycling sets . . . 127
15.2. (Open-closed) mod-minimizers . . . 128
15.3. Lower bounds for forward local schemes . . . 129
15.4. Other variants . . . 129
15.5. Toward multi-hash schemes . . . 131

16. Multiminimizers **133**
16.1. A link between density and the expected distance between selected positions . . . 133
16.2. Multiminimizers: trading time for space . . . 136
16.3. On deduplicated density . . . 137
16.4. Results . . . 140
16.5. Conclusion . . . 142

17. Sketching super-*k*-mers **145**
17.1. Preliminaries . . . 145
17.2. Fractional Hitting Sets . . . 146
17.3. Sketching technique in SuperSampler . . . 148
17.4. Results . . . 151
17.5. Conclusion . . . 155

Third discussion **157**

Conclusion & perspectives **159**

References **161**

Appendices **181**

A. A forward scheme for canonical minimizers **181**
A.1. Innermost tie-breaking . . . 181
A.2. Proof of forward property . . . 182

B. Experiments on parsing **183**
B.1. Features of the benchmarked CPUs . . . 183
B.2. Additional experiments on data read from disk . . . 183
B.3. Additional experiments on data loaded in RAM . . . 183

C. Experiments on hyper-*k*-mers **191**
C.1. Datasets . . . 191
C.2. Multi-threading efficiency of KFC . . . 191
C.3. Metagenomic benchmarks . . . 192
C.4. Effect of coverage . . . 192
C.5. Effect of not filtering unique *k*-mers . . . 192
C.6. Pangenome benchmark . . . 195

D. Proofs and experiments on multiminimizers **197**
D.1. Proof of Theorem 16.1 . . . 197
D.2. Proof of Proposition 16.1 . . . 197
D.3. Convergence of P_1 to the uniform distribution . . . 198
D.4. Monte-Carlo simulations of random minimizers . . . 198
D.5. Proof of Proposition 16.2 . . . 203
D.6. Proof of Theorem 16.2 . . . 203
D.7. Density plots for multiple *w* values . . . 207
D.8. Conservation with respect to error rate . . . 207
D.9. Linearity of indexation time . . . 207

E. Proofs and experiments on fractional hitting sets **211**
E.1. Useful lemmas . . . 211
E.2. Proof of Theorem 17.1 . . . 211
E.3. Proof of Theorem 17.2 . . . 212
E.4. Proof of Theorem 17.3 . . . 213
E.5. Proof of Theorem 17.4 . . . 214
E.6. Additional figures . . . 215

F. Useful tools **219**
F.1. Typesetting this thesis . . . 219
F.2. Bibliography management and formatting . . . 219
F.3. CLI tools I use everyday . . . 219
F.4. Rust toolchain . . . 220

Part I.

Introduction

1. Delving into genomic sequences

DNA, or deoxyribonucleic acid, is the molecule that carries the genetic information of living organisms. As you probably know, it contains four nucleotides: adenine (`A`) pairing with thymine (`T`), and cytosine (`C`) pairing with guanine (`G`). A genome is the complete set of DNA of an organism, stored across one or multiple chromosomes. The human genome, for instance, spans roughly 3.2 billion base pairs distributed across 23 chromosome pairs. It's often difficult to grasp how much information this represents, so let's try to imagine what this would amount to when printed as books. A typical novel contains roughly 80,000 words, or 500,000 characters, and is approximately 3cm wide. Thus, a single human genome would require 6400 books, enough to fill a 200m long bookshelf!

Given this amount of information, we soon realize that any kind of manual analysis would take years of effort[1], so we definitely want some form of automation. This is where sequencing technologies come into play, by turning DNA samples into a signal that we can feed to computers. However, current sequencers are not able to read entire chromosomes in one go, so the DNA is fragmented and sequenced into millions of *reads* with partial information. Moreover, sequencers regularly make errors by inserting, deleting or modifying nucleotides, resulting in inaccurate reads.

Modern sequencing technologies are divided in two main categories. *Short-read* sequencers, most notably Illumina, produce reads of 100–300 base pairs with a low cost per base and error rates below 0.5%. *Long-read* sequencers from Oxford Nanopore Technologies (ONT) and Pacific Biosciences (PacBio) generate reads of thousands to hundreds of thousands of base pairs, at the cost of higher error rates (0.5–5%) and lower throughput.

After sequencing, we're left with a large collection of noisy reads which are completely unorganized, so we'd like to correct their errors and recover their original arrangement. Going back to the book analogy, we would have many excerpts spanning a couple sentences (for short reads) or a couple pages (for long reads) that contain lots of typos. Some typos are easy to catch because they create fake words, like "sky" becoming "skt", but others are more insidious and could turn a friend into a fiend.

The process of reconstructing a sequence from a set of reads is called *assembly*. Two main strategies exist: when a closely related reference sequence is available, it can be used as a template to anchor and rearrange the reads, a process known as *reference-guided* assembly. When no such reference exists, the reads must be assembled from scratch, which is called *de novo* assembly. In practice, depending on the application, one may work either with a fully assembled genome or directly with raw reads. Both are typically stored in text-based formats: FASTA, which records a sequence identifier followed by the nucleotide sequence itself, and FASTQ, which additionally encodes a per-base quality score reflecting the sequencer's confidence at each position.

A large share of these datasets, both raw reads and assembled genomes, are saved in public archives. The most prominent ones include the Sequence Read Archive (SRA) and GenBank operated by NCBI in the US, and their European counterpart European Nucleotide Archive (ENA) operated by EMBL-EBI in the UK. Their size is growing at a staggering pace: SRA alone has surpassed 100 petabytes of data. In our book metric, this would correspond to more than 6 million km of books, enough to cover the distance from Earth to the Moon… 16 times! This exponential growth lies at the heart of the challenge we try to address: sequencing data

[1] Sanger's early sequencing work required reading gel fragments by hand, even for genomes as small as bacteriophages [SNC77].

is accumulating faster than our storage and computational capacity can keep up with. New approaches are therefore needed, ones that can scale to this ocean of data while remaining tractable on modest resources.

Throughout this thesis, we will discuss different ways of comparing sequences to one another. Historically, two broad families of methods are distinguished: *alignment-based* and *alignment-free*. Alignment-based methods usually rely on computing an edit distance between two sequences, that is the minimum number of insertions, deletions, and substitutions required to transform one into the other[2]. Their main drawback is computational cost: classical algorithms run in quadratic time [NW70; SW81], and even the more recent ones such as WFA [MMME20; Mar+23] or A*PA [GI24; Gro24] run in slightly super-linear time[3]. Alignment-free methods, by contrast, bypass the position-by-position correspondence to run in at most linear time and space. Instead, each sequence is summarized by a compact signature, most commonly derived from its k-mer content, i.e. all its substrings of fixed length k, and sequences are compared by the similarity of their signatures. We discuss the benefits of k-mer-based approaches in the next chapter, and subsequent chapters develop the data structures and algorithms needed to compare sequences at scale.

[2] In practice, repeated operations are often assigned lower costs, leading to affine-cost edit distances.
[3] For a detailed history of pairwise alignment, I recommend reading Ragnar Groot Koerkamp's thesis [Gro25a].

2. Comparing using *k*-mers

In contrast to full-text methods such as the Burrows-Wheeler Transform (BWT) [BW94; FM00], which aim to losslessly represent the original sequence while enabling more operations, we can often afford to lose some information on a sequence if this allows us to reduce its memory footprint or speed up comparisons. In this thesis, we will not cover full-text representations, and will focus on representing substrings of fixed-size k, or k-mers. In particular, we will often see sequences as *sets* of k-mers, without accounting for their positions or number of occurrences.

2.1. Coarse-grained comparisons with *k*-mers

Intuitively, indexing k-mers allows comparisons at a granularity of at least k bases, rather than at the level of individual nucleotides. This immediately speeds up comparisons by short-circuiting the first k positions, but it also means that matches shorter than k cannot be identified.

Working with fixed-size substrings naturally leads to fairly simple data structures. For small values of k (up to around 15), k-mer sets are typically dense enough in Σ^k to use a plain lookup table. For larger k, however, the k-mer set of a sequence becomes much sparser in Σ^k, and hash tables are usually preferred instead.

In the applications considered in this thesis, k typically ranges from 20 to 70, thus yielding sparse sets. Additionally, a k-mer and its reverse-complement are often treated as equivalent, since sequencing methods do not tell us which DNA strand a read originates from. We therefore designate one of the two[1] as a *canonical* representative. For this reason, k is usually restricted to odd values to avoid the degenerate case where a k-mer is its own reverse-complement[2].

2.2. De Bruijn graph representation

One immediate observation about the k-mer set of a sequence is that two k-mers at successive positions always overlap by $k-1$ bases. From this property stems the notion of *de Bruijn graph* (DBG), where each node corresponds to a k-mer and a directed edge from x to y exists whenever $x_2 \dots x_k = y_1 \dots y_{k-1}$. This definition is referred to as *node-centric*, but we can define an *edge-centric* version that's equivalent by representing k-mers as edges and $k-1$ overlaps as nodes. Figure 2.1 shows the node-centric DBG induced by the sequence `CTAAGAAGGT`, while Figure 2.2 shows its edge-centric counterpart. In this setting, any sequence of length $\geq k$ corresponds to a *walk* in the DBG. This representation can however lead to ambiguities in the presence of repeats: for instance, `CTAAGAAGGT`, `CTAAGAAGAAGGT` and `CTAAGAAGAAGAAGGT` all correspond to the same graph for $k = 3$.

Note that in practice, when a k-mer and its reverse-complement are treated as equivalent, the corresponding nodes are usually merged and DBGs become *bidirected* graphs, where each node carries two orientations and edges connect compatible ends.

2.2.1. Identifying small variations

An important property of DBGs is that they make small edits between similar sequences easy to spot [ITM12]. For instance, in Figure 2.3, substituting the last `A` with a `C` creates a *bubble*

[1] For instance, by taking the lexicographically smallest.
[2] When k is odd, the middle base of a k-mer and its reverse-complement always differ.

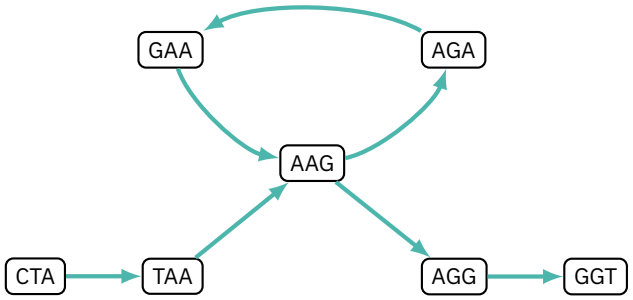


Figure 2.1.: Example of a node-centric de Bruijn graph for $k = 3$, built from the sequence `CTAAGAAGGT`.

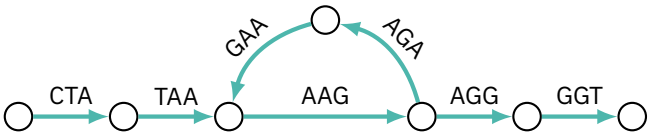


Figure 2.2.: Example of an edge-centric de Bruijn graph for $k = 3$, built from the sequence `CTAAGAAGGT`.

around the k k-mers affected by the substitution. When combined with the abundance of each k-mer, this allows erroneous k-mers in raw reads to be corrected, or different haplotypes to be distinguished within a genome. DBGs therefore prove particularly useful for genome assembly [Ban+12; KYLP19; EBC21]. Figure 2.4 shows a more complex DBG for $k = 31$ built from mRNA transcripts of SSNA1 gene, highlighting different exons identified in the graph.

Historically, this is also where de Bruijn graphs first entered bioinformatics: reconstructing a sequence from its reads raised scalability issues that motivated collapsing read depth into distinct k-mers and exploiting the compressibility of non-branching paths. Both ideas later found applications well beyond assembly.

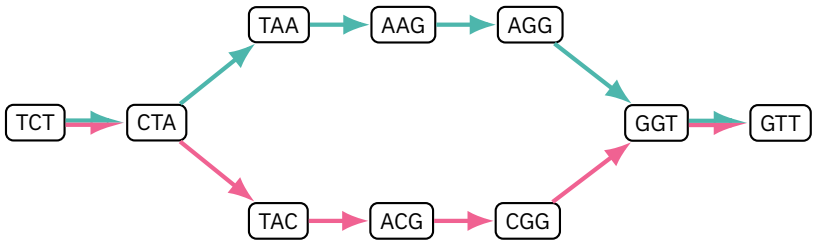


Figure 2.3.: De Bruijn graph for $k = 3$ of the sequences `TCTAAGGTT` and `TCTACGGTT`, each highlighted in a distinct color.

2.2.2. Compact representations

The set of k-mers associated to a sequence is often referred to as its k-mer spectrum[3] and denoted $\mathrm{Sp}_k(T)$ for a string T. This notion can naturally be extended to sets of strings by merging their spectra. Since having access to the k-mer spectrum is sufficient to construct the corresponding DBG, one may want to find a compact way to represent it. Rahman and Medevedev [RM21] and Břinda et al. [BBK21] formulated this problem as finding a minimum *spectrum-preserving string set* (SPSS) / *simplitigs*: given an input set of sequences $\mathcal{I}$, find a new set of sequences $\mathcal{S}$ such that $\mathrm{Sp}_k(\mathcal{I}) = \mathrm{Sp}_k(\mathcal{S})$, each k-mer appears exactly once in $\mathcal{S}$ and $\sum_{T \in \mathcal{S}} |T|$ is minimal.

[3]Depending on the context, the notion of k-mer spectrum also includes the number of occurrences of each k-mer.

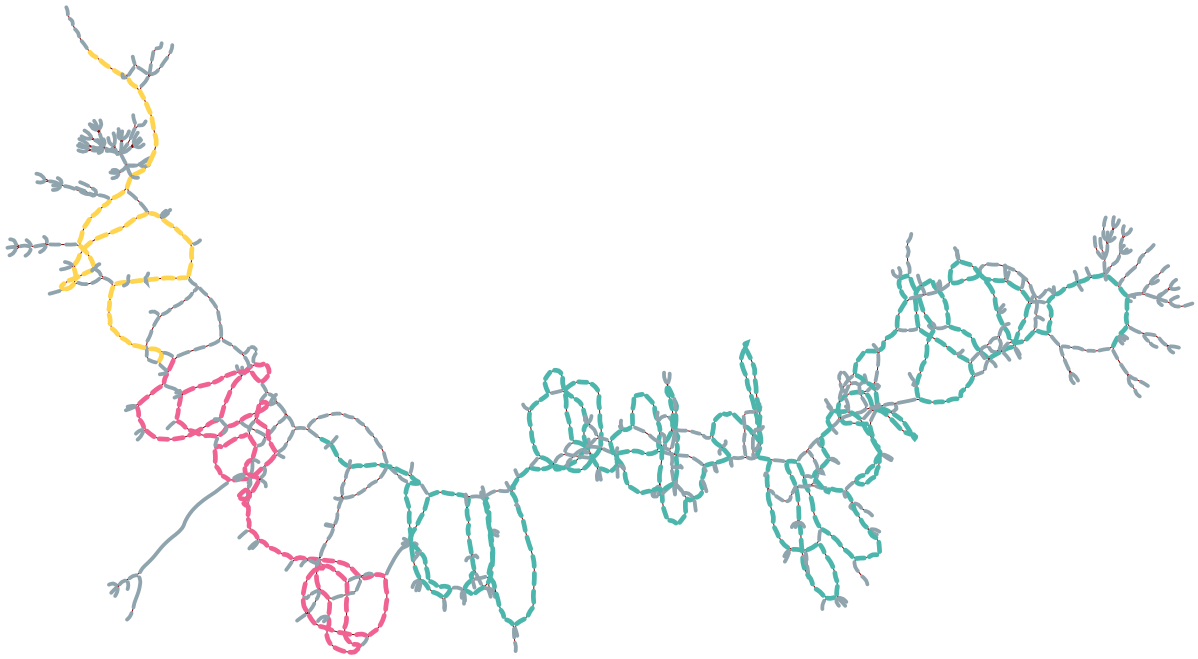


Figure 2.4.: De Bruijn graph for $k = 31$ of SSNA1 gene mRNA transcripts with 3 annotated exons, made by Bastien Degardins with Vizitig [DPM25].

A simple way to build a (non-minimal) SPSS is to decompose the edge-centric DBG into maximal paths with non-branching inner nodes, known as *unitigs* [Chi+15]. For instance, the DBG in Figure 2.2 is decomposed as $\{\mathsf{CTAA}, \mathsf{AAG}, \mathsf{AGAA}, \mathsf{AGGT}\}$. The resulting graph is called a *compacted de Bruijn graph* (cDBG), and can be computed efficiently in (external) memory [CLM16; KKDP22; CT23].

While building unitigs does not result in a minimal SPSS (unless there are no branches), Schmidt and Alanko [SA23] introduced *eulertigs* as an optimal solution based on Eulerian circuits that can be computed in linear time. Moreover, we can relax the unicity constraint and allow repeated k-mers to further reduce output size, generalizing SPSS into *repetitive SPSS* (rSPSS). Schmidt et al. [Sch+23] proposed an algorithm based on perfect matching and Eulerian circuits to find minimal rSPSS, referred to as *matchtigs*, and presented a faster greedy heuristic to approximate it. All these solutions have since been implemented as part of GGCAT [CT23].

2.3. Using *k*-mers for petabase-scale data structures

The compact representations above address how to store a k-mer set efficiently. A complementary question is how to search across very large collections of such sets. Driven by projects like Tara Oceans [Kar+11] and Pebblescout [SA24], the unprecedented growth of sequencing data [Ste+15] has motivated a parallel effort to make these massive collections efficiently searchable. The central problem here is *sequence retrieval*: given a query sequence, find all samples in the collection that contain it. In this context, k-mers provide a natural intermediate representation, as each sample can be summarized by its k-mer set and collections of such sets indexed jointly. We survey recent methods for indexing individual k-mer sets in Chapter 10, and focus on collection-level methods in the rest of this section.

A first generation of approaches relied on approximate membership data structures. Sequence Bloom Trees [SK16] organized per-sample Bloom filters in a hierarchy to prune the search space. BIGSI [Bra+19] stored these filters in a bit-sliced layout for fast intersection across hundreds of thousands of samples. REINDEER [Mar+20; Her+25] took a different angle and indexed k-mer

counts rather than their presence or absence.

Exact, lossless representations emerged with *colored compacted de Bruijn graphs* (ccDBG), which annotate each k-mer in a shared DBG with the set of samples (*colors*) containing it. The index is usually decomposed into a k-mer dictionary and a sparse annotation matrix over samples. Compressing this annotation matrix is the main algorithmic challenge: Mantis [Pan+18] groups k-mers with identical color sets into equivalence classes, MetaGraph [Kar+25] uses topology-aware compression, and Fulgor [Fan+24; CPFP24] exploits color similarity between adjacent unitigs and uses repetition-aware compression to achieve state-of-the-art index sizes.

The Logan project [Chi+24] recently proposed a complementary approach: rather than indexing raw reads directly, it first compacted the majority of the Sequence Read Archive into unitigs, drastically reducing the volume of data before indexing. Logan-Search[4] then builds a k-mer index over the resulting unitigs to enable large-scale search over all entries. Ongoing work by Rouzé et al. [RCL25] aims to further reduce the space usage of Logan's unitigs by using a compressed color-to-k-mer mapping.

[4] https://logan-search.org/

3. Sketching sequences

Sketching is the idea of summarizing a (large) dataset into a (small) set of fingerprints, called a *sketch*. A sketch is designed to be much cheaper to store and compare, while still providing good approximations for properties of interest. In this chapter, we treat sequences as sets of k-mers (without accounting for their number of occurrences or positions) and we focus on sketches approximating the *similarity* of two sequences.

3.1. Why do we want to sketch sequences?

The primary motivation to do sketching in the first place is the colossal amount of sequencing data we have to deal with. What is especially limiting is not just the length of the sequences themselves but more importantly how many of them we have to compare, for instance when comparing an unknown sample against millions of reference genomes to find the closest matches. Thus, sketching is generally used as an efficient *filtering* step, rapidly narrowing the set of sequences before applying more expensive, exact operations [Row19].

This framing naturally leads to two design goals: sketches must be *lightweight* since we may need to store millions of them, and similarity estimation must be *fast* since the number of pairwise comparisons may scale quadratically. Ultimately, both goals push against *accuracy*: a smaller sketch is cheaper to store and compare, but produces less precise estimates.

3.2. Approximating similarity

The similarity of two genomic sequences is generally defined by their *average nucleotide identity* (ANI), which measures the expected probability that a nucleotide position in a shared genomic region is identical between two genomes [KT05]. However, computing ANI exactly is quite expensive, so we typically rely on the similarity of the corresponding k-mer sets instead and use it to derive an approximation of ANI.

To measure the similarity of two sets A and B, we generally use the *Jaccard index* (sometimes called Tanimoto coefficient) defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Nevertheless, we still want to avoid having to intersect large k-mer sets. Thus, instead of working with the original sets, we sample them into smaller sketches $S(A)$ and $S(B)$ for faster comparison. Figure 3.1 illustrates this idea for "bottom MinHash" sketching that we present in the next section.

3.3. MinHash and its variations

Now that we know we want to sketch k-mer sets to quickly estimate their Jaccard similarity, the main question becomes: how do we design sketching methods that are as accurate as possible with a tight memory budget?

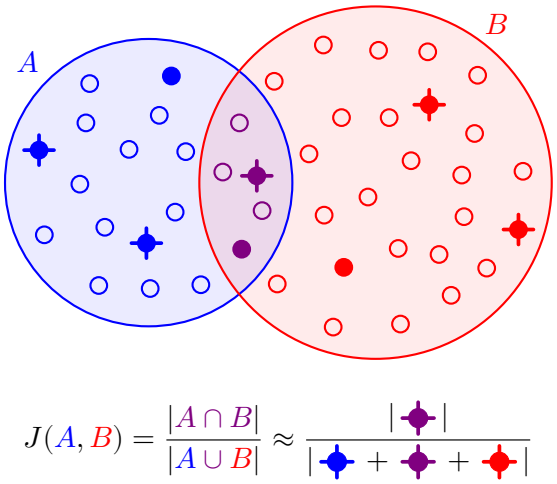


Figure 3.1.: Overview of Jaccard similarity and how bottom MinHash can approximate it, adapted from Ondov et al. [Ond+16]. Dots filled in blue represent the sketch of A, while those filled in red represent the sketch of B and crosses indicate the sketch of $A \cup B$.

3.3.1. MinHash

MinHash [Bro97] is one of the simplest solutions to this problem. Given a regular hash function h, it computes the sketch of a set A as

$$h(A) = \min_{x \in A} h(x)$$

The nice property of this definition is that the probability for the minimum hashes of two sets to be equal exactly matches their Jaccard index, in other words it is an *unbiased estimator*:

$$\mathbb{P}(h(A) = h(B)) = J(A, B)$$

This can be modelled as a Bernoulli variable with a success probability of $j = J(A, B)$, and a variance equal to $j(1 - j)$.

While comparing a single hash gives the desired result on expectation, this estimate has a high variance. Thus, in order to reduce the variance, we want to average multiple independent samples. One way to do this is to generate s independent hash functions $h_1, \dots, h_s$ by using multiple seeds:

$$S(A) = (\min_{x \in A} h_1(x), \dots, \min_{x \in A} h_s(x)) \widehat{J}(A, B) = \frac{1}{s} \sum_{i=1}^{s} \mathbf{1}_{h_i(A) = h_i(B)}$$

This estimator is still unbiased and its variance is equal to $j(1 - j)/s$, leading to standard deviation proportional to $1/\sqrt{s}$.

3.3.2. Bottom MinHash

One drawback of using s hash functions is that building the sketch takes $\mathcal{O}(Ns)$ time for a sequence of size N. What we can do instead is to select the s smallest hashes using a single hash function, which we refer to as *bottom-s MinHash*.

A common way to maintain the s smallest values is to use a max-heap, only updating it if the incoming hash is smaller than the current max of the sketch. While the construction time can be upper-bounded by $\mathcal{O}(N \log s)$, Broder [Bro97] refined the bound down to $\mathcal{O}(N + s \log N \log s)$ by noting that the heap is only updated $\mathcal{O}(s \log N)$ times on average.

Broder [Bro97] proved that this sketch also provides an unbiased estimator of Jaccard similarity[1]

$$\widehat{J}(A,B) = \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$$

where $S(A)$ denotes the set of the s smallest hashes in A. This approach has been used in Mash [Ond+16] to define a distance approximating ANI for genomes and metagenomes, and is illustrated in Figure 3.1.

3.3.3. FracMinHash

One limitation of fixed-sized sketches like (bottom) MinHash is that they perform poorly on sets of very dissimilar sizes [Row19]. To address this limitation, we can create a sketch whose size is proportional to the k-mer set. Given a threshold $t \in [0,1]$, and assuming the hash values are in $[0,1]$, we select any hash smaller than t:

$$S_t(A) = \{h(x) : x \in A, h(x) < t\}$$

This method is now known in the literature as FracMinHash [Irb+22], but has appeared under different names such as scaled MinHash [Pie+19], universe minimizers [EBC21] or mincode submer [Edg21] in the past.

Given a sequence of size N, and the corresponding k-mer set of size n, a FracMinHash sketch contains nt elements in expectation and can be built in $\mathcal{O}(N)$ time with a linear scan on the hash values. This approach has the benefit of being simple and efficient, and supports straightforward set operations such as union or intersection (for instance $S_t(A \cup B) = S_t(A) \cup S_t(B)$).

Rahman Hera et al. [RPK23] showed that the naive estimator $J(S_t(A), S_t(B))$ is biased, and that it can be corrected as follows:

$$\widehat{J}(A,B) = \frac{J(S_t(A), S_t(B))}{1-(1-t)^{|A \cup B|}}$$

However, one may notice that this formula depends on $|A \cup B|$, which is precisely what we were trying to avoid with $J(A,B)$ in the first place. As a workaround, Rahman Hera et al. [RPK23] suggested to approximate $|A \cup B|$ by $|S_t(A \cup B)|/t$, leading to the following approximation:

$$\widehat{J}(A,B) = \frac{J(S_t(A), S_t(B))}{1-(1-t)^{|S_t(A) \cup S_t(B)|/t}}$$

Among practical use cases, FracMinHash is implemented as part of the popular sourmash toolchain [Pie+19; Irb+22] and has been used to index all public metagenomes from the Sequence Read Archive [IPB22].

3.4. Partitioned sketches and smaller fingerprints

3.4.1. Partitioned sketches

An intuitive idea to increase the number of samples without using multiple hash functions is to partition the universe of hash values into s parts, and compute a sketch for each of them. This technique is sometimes referred to as *one permutation hashing* in the literature [LOZ12].

A simple way to implement this is to set $s = 2^p$ and use the first p bits of a hash to select its partition. This has three main benefits: 1/ this is efficient to compute with bitwise operations, 2/ this allows us to save p bits for each element of the sketch since the first p bits are

[1] Note that $J(S(A), S(B))$ is *not* a correct estimator.

implicitly encoded by the partition and 3/ this enables faster (and parallelizable) comparisons at partition-level.

One limitation of using many partitions is that some of them may end up being empty if the input is not large enough. Some articles addressed this issue by designing *densification* strategies [SL14]: each empty slot is filled with other values from the sketch. Note that densification is not mandatory and that some sketching schemes simply skip empty slots during comparison, at the cost of a slightly worse precision [YW20].

3.4.2. *b*-bit MinHash

One thing we may notice when computing min hashes is that the low bits carry much more information than the high ones (which are more likely to be zeros). Based on this observation, we can restrict the selected hashes to their lowest b bits to save space [LK10]. However, this introduces a collision probability which must be taken into account to correct the estimator: two distinct hashes might share the same lowest b bits, which would lead to an overestimated similarity. This technique is used in tools such as BinDash [Zha19] to estimate genome distance.

3.4.3. HyperMinHash

Yu and Weber [YW20] extended this idea with an additional trick: instead of discarding the high bits, use them in a *HyperLogLog* counter [FFGM07]. Each hash is thus represented by both its number of leading zeros and the b bits that follow this run of zeros (skipping the first one). Compared to b-bit hashing, HyperMinHash has the benefit of being updatable after construction and provides a precise cardinality estimator with HyperLogLog. This technique inspired the Dashing [BL19; BL23] toolkit.

3.4.4. MaxGeomHash

MaxGeomHash [HKM25] is a recent method that draws inspiration from many of the previous techniques to design a sketch that scales sublinearly with the input size. Its main idea is to partition the hash values by their number of leading zeros instead of their first p bits, and select the largest[2] s hash values of each partition. For a fixed s, this leads to a sketch of $\mathcal{O}(\log n)$ elements.

Moreover, Hera et al. [HKM25] also introduced a variant called α-MaxGeomHash, which changes the number of elements selected in each partition: given $\alpha \in (0, 1)$, the partition with i leading zeros will select $2^{i\frac{\alpha}{1-\alpha}}$ elements to correct the imbalance of partition sizes. With this change, the size of the sketch becomes $\mathcal{O}(n^\alpha)$, for any desired $\alpha \in (0, 1)$.

These two methods offer a middleground between fixed-sized sketches such as (bottom) MinHash and linear ones such as FracMinHash, while providing similar unbiased estimators for Jaccard similarity:

$$\widehat{J}(A, B) = \frac{\sum_{i=1}^{\log n} |S_i(A \cup B) \cap S_i(A) \cap S_i(B)|}{\sum_{i=1}^{\log n} |S_i(A \cup B)|}$$

3.5. Other types of sketches

The sketching methods that we presented so far all have in common that they sample k-mer hashes to estimate Jaccard similarity, but many other families of sketches have been designed with different goals in mind [Che+26]. In this section, I will cover some that I find especially promising.

[2] Using the smallest hashes would work just as well.

3.5.1. DotHash for intersection size

DotHash [Nun+23] takes a different approach than MinHash-like sketches: rather than sampling a subset of k-mers and storing their hash, it computes a random projection for every k-mer and aggregates *all of them* in a vector of fixed dimension d. Note that d is independent of the size n of the k-mer set but the precision required for each aggregated component increases with n.

A key property of DotHash is that it can efficiently estimate the intersection size of two sets A and B by computing the dot product of the corresponding sketch vectors. In particular, by computing the dot product of the vector of A with itself, we obtain an estimator of the cardinality of A. Therefore, Jaccard similarity can be estimated as

$$\widehat{J}(A,B) = \frac{S(A)\cdot S(B)}{S(A)\cdot S(A) + S(B)\cdot S(B) - S(A)\cdot S(B)}$$

However, DotHash has a major drawback: it only works on deduplicated k-mer sets and cannot easily detect duplicates since the vectors are aggregated. In particular, if a given k-mer has c_A occurrences in A and c_B in B, it will be counted $c_A \times c_B$ times in their dot product, leading to an overestimated intersection size. Thus, DotHash is suitable on repeat-free sequences such as unitigs, but cannot be directly used on arbitrary sequences.

HyperGen [Xu+24] addresses this limitation by combining DotHash with FracMinHash sampling to sparsify and deduplicate its input. This approach results in a better space-accuracy tradeoff compared to existing MinHash-based tools, and is suitable for hardware acceleration.

Ongoing work by Faure et al. [Fau+25] aims to adapt DotHash for large-scale all-vs-all comparisons of metagenomes.

3.5.2. SubseqSketch for edit distance

Recent results by Chen et al. [CPS25] focus on estimating the edit distance between two sequences without having to use an expensive alignment algorithm. Their approach, named SubseqSketch, relies on (non-contiguous) *subsequences* rather than k-mers, and keeps track of their longest prefix match with respect to multiple random references. The match lengths are then used to compute a cosine similarity that strongly correlates with edit similarity. However, while this approach results in an improved sensitivity, it remains more expensive to compute than simple scans such as FracMinHash.

3.6. Toward locality-aware sampling

The sketching methods presented in this chapter share the property that each k-mer is selected or not independently of its neighbors in the sequence. This context-free property makes them well suited for global similarity estimation, where all that matters is the overall composition of the k-mer set. For tasks that instead demand a guarantee that two sequences sharing a long local match will also share a selected k-mer, such as read mapping, the next chapter turns to a different family of sampling strategies.

4. Sampling with minimizers

The sketching methods presented in Chapter 3 compress a sequence into a compact summary, but they treat the sequence as a simple "bag" of k-mers: two sequences with a high Jaccard index are considered similar regardless of *where* the shared k-mers appear.

Many bioinformatics tasks require something stronger: a guarantee that two sequences sharing a long local exact match will also share at least one selected k-mer at corresponding positions. This *locality* property is what distinguishes minimizers from general "context-free" sketching: here selected k-mers act as representatives of their local context.

4.1. Definition and fundamental properties

The concept of minimizers was introduced independently by two groups in 2003–2004. Schleimer et al. [SWA03] introduced *winnowing* in the context of document fingerprinting and plagiarism detection: a sliding window scans the document and the smallest k-mer of each window (according to some ordering) is selected as a fingerprint. Roberts et al. [Rob+04] proposed the same idea for biological sequences, introducing the term *minimizer*, with the explicit goal of providing a locality-preserving sketch: any two sequences sharing a long exact match are guaranteed to share at least one minimizer.

A minimizer scheme is parameterized by three values: the k-mer length k, the window size w, and a total ordering $\mathcal{O}_k$ on k-mers. A *window* of size w is a set of w consecutive k-mers, spanning a substring of length $\ell = w + k - 1$. The *minimizer* of a window is the k-mer that is smallest according to $\mathcal{O}_k$, with ties broken by selecting the leftmost occurrence[1]. Sliding the window one position at a time along the sequence yields the *minimizer sketch* of that sequence: the (deduplicated) sequence of minimizer positions. We discuss efficient implementations of this sliding window algorithm in more detail in Chapter 8.

An essential property of minimizers is the so-called *window guarantee*.

Property 4.1 (Window guarantee)**.** Any window of w consecutive k-mers, i.e. any substring of length $\ell = w + k - 1$, selects one minimizer.

This property has two important consequences that make minimizers especially useful in practice.

Property 4.2 (Conservation)**.** Any pair of sequences sharing an exact match of length at least $\ell = w + k - 1$ will share at least one minimizer.

Property 4.3 (Bounded distance)**.** The distance between consecutive minimizers is at most w.

These two properties together ensure that minimizers can serve as *anchors* for sequence comparison: shared minimizers between a query and a reference indicate candidate matching positions, and the bounded distance guarantees that no long match is missed [Ndi+24].

4.2. Density and orderings

Definition 4.1 (Density of a minimizer scheme)**.** The *density* of a minimizer scheme is the expected fraction of k-mers selected over all positions of a random sequence.

[1] Choosing the leftmost occurrence is a convention, some schemes use the rightmost one or select all tied k-mers.

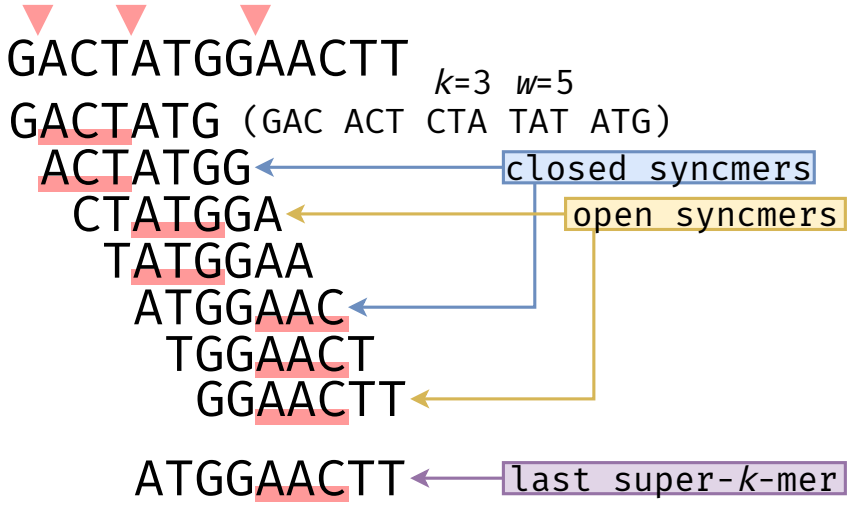


Figure 4.1.: Example of lexicographic minimizer selection, for $k = 3$ and $w = 5$. Each line corresponds to a sliding window of 5 consecutive 3-mers, selecting a minimizer highlighted in red. The starting positions of the 3 minimizers are marked by red arrows above the sequence. Consecutive windows selecting the same minimizer position can be merged into a single super-k-mer.

Since consecutive minimizers are at most w positions apart (Property 4.3), the density is always lower-bounded by $1/w$. We'll see in the following subsections that the ordering used to compare k-mers can have a big impact on the density, and thus on the size of the resulting sketch.

4.2.1. Lexicographic ordering

The simplest ordering we can use is the lexicographic order on k-mer strings. While it requires no precomputation and is straightforward to implement, lexicographic ordering has well-known limitations: homopolymers (especially repeated `A`s) tend to select adjacent k-mers, thus increasing the density and skewing the distribution of minimizer values [Mar+17; ZMK23; IMS24]. In particular, Marçais et al. [Mar+17] report an empirical density of $2.24/(w+1)$ for lexicographic minimizers with $k = 10$ and $w = 10$. An example of lexicographic minimizer selection is given in Figure 4.1.

4.2.2. Random ordering

Another popular solution is to order k-mers randomly, typically using a pseudo-random hash function. Randomizing the order reduces the impact of poly-`A`s and spreads out the distribution of minimizer values [ILMS26]. Zheng et al. [ZKM20] proved that random minimizers achieve a density of $2/(w+1) + o(1/w)$, a ~10% improvement over lexicographic density, as long as $k > 3\log_{|\Sigma|}(w+1)$. In practice this condition is almost always satisfied and random ordering is the default choice in most bioinformatics tools.

This gap between the density of $2/(w+1)$ and the $1/w$ lower bound motivated the design of improved orderings and low-density schemes, which we discuss in more details in Chapter 15.

4.3. Super-k-mers

An important property of minimizer schemes is that consecutive windows tend to share the same minimizer. More precisely, random minimizers are shared between $(w+1)/2$ windows on average

(i.e. the inverse of the density). Therefore, instead of being represented independently, the windows associated to a given minimizer occurrence can be merged into a compact representation called super-k-mer [Li+13]. Formally, a super-k-mer is the smallest substring covering all windows selecting a given minimizer position. An example of super-k-mer is depicted in Figure 4.1.

> **Different notations**
>
> In the context of super-k-mers, the minimizer size is usually denoted m (instead of k previously) while k denotes the number of bases in a window (previously denoted as ℓ), i.e. $k = w + m - 1$.

Super-k-mers have been widely adopted in bioinformatics tools for two main reasons. First, storing super-k-mers is much more compact than storing each k-mer individually (by avoiding duplicated overlaps). We will see in Chapter 14 that transforming a sequence into a set of super-k-mers roughly increases its space usage by a factor 3 (while individual k-mers lead to a factor k) and that we can reduce it further with hyper-k-mers. Second, since each k-mer belongs to a single super-k-mer, this induces a natural partition of the original sequence. This partitioning method is useful to distribute k-mers in memory and balance the computational load, and preserves locality by grouping consecutive k-mers sharing the same minimizer.

4.4. Improving conservation

While we have seen that density directly impacts the space usage of minimizers and super-k-mers, *conservation* is an important complementary metric that measures how well minimizers are preserved under mutations. Formally, conservation is defined as the expected fraction of bases covered when matching selected k-mers from the original sequence to those in the mutated version [Edg21; SY21].

4.4.1. Syncmers

Syncmers were introduced by Edgar [Edg21] as an alternative to minimizers that improve conservation. Rather than selecting the minimizer of each window, syncmers select the *window itself* if its minimizer is located at a specific position within the window. In *closed syncmers*, the minimizer must appear at the first or last position of the window. In *open syncmers*, it must appear at a fixed interior position, typically the middle one (assuming an odd number of elements). Examples of open and closed syncmers are shown in Figure 4.1.

> **Different notations**
>
> In the context of syncmers, the minimizer size is usually denoted s (instead of k previously) while k denotes the number of bases in a window (previously denoted as ℓ), i.e. $k = w+s-1$.

Closed syncmers are known to have a density of $\frac{2}{k-s+1} = \frac{2}{w}$ and a window guarantee of $k - s = w - 1$ [Edg21; SY21]: any window of $w - 1$ consecutive k-mers (i.e. $2w - 2$ consecutive s-mers) contains at least one closed syncmer. Open syncmers, on the other hand, achieve half the density at $\frac{1}{k-s+1} = \frac{1}{w}$, but have no window guarantee [SY21].

The distinction between selecting a subword versus selecting a window gives minimizers and syncmers different properties and use cases. While a minimizer depends on the context around it and may change if another k-mer in the window becomes smaller after a mutation, a syncmer is guaranteed to stay selected even if the context around it is altered (we say that it is *1-local*). Because of this, syncmers achieve a higher conservation than random minimizers, with open syncmers performing better than closed syncmers, making them a better option for anchor

selection in mappers [SY21]. On the other hand, since each window selects a unique minimizer, this minimizer can serve as a representative of the whole window for partitioning, while syncmers do not serve the same role.

4.4.2. Strobemers

Strobemers were introduced by Sahlin [Sah21] to address a different limitation of k-mer-based sampling: a single mutation invalidates k consecutive k-mers, preventing any match within these $2k-1$ bases. Instead of selecting a single contiguous k-mer, strobemers link two or more shorter k-mers (called *strobes*) sampled at variable offsets to produce a spaced k-mer. In particular, this spaced k-mer can still be selected if a mutation occurs in the gaps between the strobes, thus making it more robust. Strobemers have been shown to improve sensitivity in read alignment, and are the basis of the strobealign aligner [Sah22].

4.5. Practical use cases

A careful reader may notice that it took more than a decade after the introduction of minimizers for them to appear in widely-used tools [WS14; Chi+15; Li16]. This delay is not entirely coincidental, and actually follows the development of third-generation sequencing that made long reads widely available. Minimizers are particularly well-suited for long reads, where k-mer sampling is essential to reduce the computational cost, and it was only once such data became accessible that minimizers reached their full practical potential. In the following subsections, we briefly survey the main use cases of minimizers in practical applications.

4.5.1. Minimizers as anchors

One of the main applications of minimizers is read mapping, with the *seed-chain-align* paradigm popularized by minimap [Li16; Li18; SBCM23]. In this paradigm, both the reference and the query use minimizers as *seeds*, and index them along with their positions. Matching minimizers are then identified as *anchors*, indicating potentially large matches, and these anchors are *chained* using dynamic programming. The best chain of anchors is finally *extended* into a complete alignment, reducing the quadratic cost of a single large alignment to a more efficient succession of short alignments.

4.5.2. Minimizers for partitioning

Another important use of minimizers is that they can act as a representative of their window, which is useful to distribute k-mers in memory. This property is used for building compacted de Bruijn graphs [Chi+15; HM20; CT23] or counting k-mers [LY15; DKGD15; KDD17], where each partition corresponds to a given minimizer or subset of minimizers.

4.5.3. Minimizers as lossy fingerprints

While minimizers are often used as part of a bigger exhaustive index, they can also be used on their own as a lossy sketch of the original sequence. For instance, Kraken2 [WLL19] classifies reads against a reference database by mapping each minimizer to a lowest common ancestor in a phylogenetic tree. While this is less accurate than having this information for all k-mers, storing only minimizers reduces the database size by 85% compared to the original Kraken [WS14]. Needle [Dar+22] uses the same trick to accelerate transcript quantification, while Kaminari [Lev+26] uses it to build a lightweight colored de Bruijn graph.

4.5.4. Minimizer-space data structures

Finally, another idea that emerged recently is to transform the original sequence into its sequence of minimizers, making it shorter but with a larger alphabet, and work directly in this *minimizer space*. Ekim et al. [EBC21] successfully applied this technique to accelerate genome assembly by building a minimizer-space de Bruijn graph, where k-min-mers correspond to a chain of k consecutive minimizers. Ekim et al. [Eki+23] later used the same technique to speed up long read mapping. It should be noted, however, that the term "minimizer" used in these two contributions does not correspond to our definition and is actually closer to a FracMinHash sketch (presented in Chapter 3). Ayad et al. [Aya+25] generalized this idea to build other string data structures in sketch space, illustrating the approach with a minimizer-space suffix array that improves space efficiency.

Part II.

High-performance sequence processing

All we have to decide is what to do with the cycles that are given us.

—not J. R. R. Tolkien

In the introduction we established that some genomic pipelines routinely process hundreds of terabases of data. At that scale, even a constant-factor improvement in throughput translates directly into wall-clock time. This part asks how close to the hardware limit a genomic pipeline can actually run and what it takes to get there. Chapter 5 introduces the vectorization model and the SIMD instruction sets available on common hardware. Chapters 6, 7 and 8 develop vectorized algorithms for the three core primitives of any genomic pipeline: parsing FASTA/FASTQ input, computing rolling hashes over k-mers, and extracting minimizers from a sequence. Chapter 9 applies these primitives to sequence filtering.

5. A primer on vectorization

CPU performance scaled remarkably through the late 20$^{\text{th}}$ century, driven by increasing transistor counts (roughly following the famous *Moore's law*) and rising clock frequencies. However, as the clock frequencies started facing physical limits, CPU manufacturers invested in new solutions to keep improving the computation throughput. The main trend over the last decades has been to focus on handling more operations in parallel, either by adding more CPU cores (*task-level* parallelism), or by increasing the number of operations done by each core (*instruction-level* and *data-level* parallelism). In particular, many applications such as scientific computing, signal processing or regex engines benefit especially from data-level parallelism, where the same operation is applied *independently on multiple data* [HPK26]. In this chapter, we will present how vectorized instructions can speed up these use cases.

5.1. A brief history of vectorized extensions

The concept of applying a single instruction to multiple data elements simultaneously (formally known as SIMD) predates the personal computer era. The Cray-1 [Rus78], designed by Seymour Cray and introduced in 1976, was one of the first successful vector processors. It exposed 8 vector registers of 64 double-precision floats each, with dedicated pipelined units that could operate on entire registers in a single instruction. This made it particularly well-suited to the dense numerical workloads of scientific computing, but its vector model stayed confined to high-performance computing for the following decades. The landscape shifted in 1997 when Intel introduced the MMX extension to consumer CPUs, motivated by the growing demands of multimedia workloads [PWW97]. MMX exposed 64-bit registers holding multiple packed integers, establishing the template of the extensions that followed. The main SIMD instruction set extensions available on modern x86 and ARM CPUs are summarized in Table 5.1 and Table 5.2 respectively.

Table 5.1.: Main x86 SIMD instruction set extensions [Int26], with desktop adoption from the Steam Hardware Survey [Val26][1].

Extension	Year	Width	Key operations	Adoption
SSE	1999	128 bits	Packed single-precision float	~100%
SSE2	2001	128 bits	Packed double-precision float + integers	98.09%
SSE3	2004	128 bits	Horizontal add, duplicate loads	98.09%
SSSE3	2006	128 bits	Byte shuffle, abs, sign	98.01%
SSE4.1	2007	128 bits	Blend, dot product	97.97%
SSE4.2	2008	128 bits	String compare, popcount	97.93%
AVX	2011	256 bits	256-bit float operations	96.93%
AVX2	2013	256 bits	256-bit integers, permute, gather	94.84%
AVX-512[2]	2016[3]	512 bits	512-bit float + int, mask registers, scatter	21.89%[4]

Table 5.2.: Main ARM SIMD instruction set extensions [Arm26].

Extension	Year	Width	Key operations	Adoption
NEON	2007	128 bits	Packed float + integers	All
SVE	2016	128–2048 bits[5]	Vector-length agnostic operations, per-element masking, gather/scatter	HPC/server
SVE2	2019	128–2048 bits	Complex arithmetic, bit deposit/extract	HPC/server

5.2. SIMD vectorization in practice

All the SIMD extensions described above (apart from SVE/SVE2) follow the same design: each wide register is divided into equally-sized *lanes* of 8–64 bits (e.g. 32 lanes of 8 bits packed in a 256-bit register), and each instruction operates independently on every lane in parallel. Supported operations include arithmetic (excluding integer division and modulo), bitwise operations and comparisons. Listing 5.1 and Listing 5.2 show an example computing the sum of 32-bit signed integers in AVX2 and NEON respectively.

This paradigm enforces two main constraints on algorithm design. First, there is no branching on individual lane values: the usual workaround is to compute a boolean mask for each lane and use it to *blend* the results of two paths[6]. Second, instructions that combine values across lanes (known as *cross-lane* operations) are rare: the main ones are horizontal reductions and permutations.

[2] AVX-512 is a family of sub-extensions. All modern implementations ship F (foundation), BW (byte/word operations), DQ (doubleword/quadword), and VL (128/256-bit variants of AVX-512 instructions) together. The Steam survey confirms identical adoption across all four.

[3] First implemented in 2016 for servers and 2017 for desktop, then completely removed from Intel desktop processors in 2022 (Alder Lake and its successors). AMD implemented it in 2022 (Zen 4).

[4] Apart from recent AMD CPUs, AVX-512 is only available for servers, thus explaining the low adoption for consumer CPUs.

[1] I was not expecting to cite Steam in my thesis, but here I am.

[5] While the SVE specification supports up to 2048 bits in theory, it has only been implemented for up to 512 bits as of today.

[6] I recommend reading https://en.algorithmica.org/hpc/simd/masking/ for a deeper introduction.

Listing 5.1 Sum of `i32` in AVX2 using 256-bit registers.

```
let mut sum = _mm256_setzero_si256(); // initialize with 0s
for chunk in slice.chunks(8) { // for each chunk of 8 32-bit ints
    let vec = _mm256_loadu_si256(chunk.as_ptr()); // load in a 256-bit register
    sum = _mm256_add_epi32(sum, vec); // add each lane to the existing sum
}
let mut res = [0i32; 8];
_mm256_storeu_si256(res.as_mut_ptr(), sum);
return res.iter().sum::<i32>(); // manual horizontal sum of the lanes
```

Listing 5.2 Sum of `i32` in NEON using 128-bit registers.

```
let mut sum = vdupq_n_s32(0); // initialize with 0s
for chunk in slice.chunks(4) { // for each chunk of 4 32-bit ints
    let vec = vld1q_s32(chunk.as_ptr()); // load in a 128-bit register
    sum = vaddq_s32(sum, vec); // add each lane to the existing sum
}
return vaddvq_s32(sum); // horizontal sum of the lanes
```

One of the main drawbacks of hand-written SIMD code as shown in Listing 5.1 and Listing 5.2 is that it requires a specific implementation for each SIMD extension that we want to support, making it difficult to maintain and reducing portability. However, trusting the compiler to auto-vectorize our code is not a reliable option either since it can easily break with a minor change, a new compiler version or a different architecture. A popular alternative is to use portable SIMD libraries, such as Rust's portable_simd or wide[7], which provide a unified API that is compiled to the appropriate SIMD extension. An example of a portable SIMD implementation (supporting both x86 and ARM CPUs) is given in Listing 5.3.

Listing 5.3 Sum of `i32` using `portable_simd`.

```
#![feature(portable_simd)]
use std::simd::Simd;

let mut sum = Simd<i32, 8>::splat(0); // initialize with 0s
for chunk in slice.chunks(8) { // for each chunk of 8 32-bit ints
    let vec = Simd<i32, 8>::from_slice(chunk); // load in SIMD register
    sum += vec; // add each lane to the existing sum
}
return sum.reduce_sum(); // horizontal sum of the lanes
```

5.3. The example of memchr

Another instructive example is `memchr`, which finds the first occurrence of a given byte in a buffer. A naive scalar implementation, detailed in Listing 5.4, reads one byte per iteration, giving a throughput of one byte per cycle at best.

[7] At the moment, `wide` is less complete than `portable_simd` but has the benefit of being stable.

Listing 5.4 Scalar implementation of `memchr`.

```rust
fn memchr_scalar(buf: &[u8], needle: u8) -> Option<usize> {
    for (i, &byte) in buf.iter().enumerate() {
        if byte == needle { return Some(i); }
    }
    return None;
}
```

On the other hand, an AVX2 implementation, detailed in Listing 5.5, processes 32 bytes per iteration, following a three-step pattern that recurs in many applications. First, the needle byte is broadcast across all 32 lanes of a 256-bit register. Second, 32 bytes are loaded from the buffer and compared against the broadcast register in a single instruction, producing a boolean mask for equality. Third, the vector mask is collapsed into a scalar bitmask using `movemask`, which extracts the most significant bit of each lane in a SIMD register into a 32-bit integer (i.e. one bit per lane). The position of the first match is then found by counting trailing zeros in the resulting bitmask.

Listing 5.5 AVX2 implementation of `memchr` (type conversions and `unsafe` keywords omitted for clarity).

```rust
fn memchr_avx2(buf: &[u8], needle: u8) -> Option<usize> {
    let needles = _mm256_set1_epi8(needle); // broadcast to all 32 lanes
    let mut i = 0;
    while i + 32 <= buf.len() {
        let chunk = _mm256_loadu_si256(buf.as_ptr().add(i));
        let cmp   = _mm256_cmpeq_epi8(chunk, needles); // compare all 32 lanes
        let mask  = _mm256_movemask_epi8(cmp);  // 1 bit per matching lane
        if mask != 0 {
            return Some(i + mask.trailing_zeros()); // index of first match
        }
        i += 32;
    }
    // handle the <32 remaining bytes manually
    return buf[i..].iter().position(|&b| b == needle).map(|j| i + j);
}
```

The performance difference is substantial, approaching a 32× speedup on long strings, which is precisely why vectorized primitives should be preferred. The broadcast-compare-scan pattern illustrated here is not specific to `memchr`. It appears whenever we need to locate a particular value in a stream of data, which we will extensively use for parsing in the next chapter.

6. Vectorized sequence parsing

> **Note**
>
> This chapter is adapted from Martayan et al. (2026), accepted to RECOMB-Arch 2026 and to be published.

Sequence bioinformatics has always been closely tied to stringology and compression theory, but connections to algebraic automata have remained comparatively rare. Recent progress has opened new paths toward high-performance parsers that are both formally grounded and practically competitive. This chapter explores one such connection, applying these techniques to the FASTA and FASTQ formats.

The central contribution of this chapter consists of new vectorized algorithms for high-throughput FASTA/Q parsing, which support on-the-fly detection and handling of non-ACTG characters, and provide bitpacked representations of the sequence. Formally, given an input FASTA/Q file, we produce a structured iterator over the (possibly bitpacked) content of each record. This approach is implemented in *Helicase*, a Rust library providing a configurable interface to expose caller-requested fields for each record, avoiding unnecessary computation. We evaluate our approach on both x86 and ARM architectures. Across all tested scenarios, Helicase matches or exceeds the throughput of existing single-threaded parsers while maintaining feature parity. In addition, it provides extended functionality, such as bitpacked sequence output, without compromising performance. For data already loaded in RAM, Helicase is able to parse uncompressed short reads at 27 GB/s and long reads at 49 GB/s on an Apple M3 Pro using a single thread, reaching the core memory bandwidth. Our library is available at https://github.com/imartayan/helicase under a permissive open-source license.

6.1. Background

6.1.1. Description of the formats

FASTA and FASTQ are text-based formats containing a sequence of *records*.

FASTA records contain two fields: (1) a single-line header starting with `>` followed by an optional description and (2) a multi-line ASCII-encoded sequence. The ASCII-encoded sequence contains `A/C/T/G` for DNA and other IUPAC codes to indicate ambiguous bases, and can be split into an arbitrary number of lines until the next record. A FASTA record can be matched with the following regular expression:

$$\underbrace{\texttt{(>[\^{}\textbackslash n]*\textbackslash n)}}_{\text{header}}\underbrace{\texttt{(?:[\^{}>]*\textbackslash n)+)}}_{\text{sequence}} \tag{6.1}$$

FASTQ records contain four fields: (1) a single-line header starting with `@` followed by an identifier and an optional description, (2) a single-line ASCII-encoded sequence, (3) a single-line separator starting with `+` optionally followed by an identifier and (4) a quality line of the same length of the sequence encoding the quality score of each base at the corresponding position. Compared to FASTA, the ASCII-encoded sequence also contains `A/C/T/G` and other IUPAC codes, but cannot be split into multiple lines. The quality line, however, may contain *any character* in the ASCII range [33, 126] including `@`, `+` or any alphabetic character. Also note that,

while the quality represents almost 50% of the content of a FASTQ file, it is often discarded during the analysis.

6.1.2. Efficient parsing

At its core, parsing FASTA and FASTQ files reduces to a lexical analysis problem over large byte streams. The parser must identify structural delimiters, such as line feeds (`\n`), header markers (`>` and `@`), and separators (`+`) and segment the input into records accordingly. Since modern datasets often reach billions of reads, the dominant cost is no longer algorithmic complexity but the raw throughput at which these critical positions can be detected.

Reaching a high throughput requires minimizing branching and favoring regular, predictable computation patterns that can be efficiently executed on wide data paths. This constraint naturally shifts the focus from traditional parsing strategies toward approaches that process the input stream in bulk while limiting branches. Therefore, using vectorization techniques (introduced in Chapter 5) seems essential to achieve this goal. Moreover, since parsing FASTA/Q requires searching for *multiple* markers simultaneously, we will see that relying on multiple calls to built-in functions like `memchr` does not fully exploit hardware capabilities.

6.1.3. Existing work

Most state-of-the-art parsers share a pretty similar design. The input file is first split into a sequence of lines using `memchr` to locate line feeds (`\n`). For FASTQ files, counting the lines is sufficient for the parsing logic since each field spans over a single line. For FASTA files, however, we also need to check the first character of the line to determine if it starts a header (`>`). This usually incurs a second check after each `memchr` match to detect a new record. In particular, both `kseq` [Li09] and `needletail` [One19], two of the most popular parsers in the community, rely on this approach. According to Heng Li's `biofast` benchmark [Li20], `needletail` is the fastest single-threaded parser currently available, so this will be the baseline we compare against.

Recent methods focused more specifically on parallelization strategies for parsing [Zha+23] and decompressing [Pat+25] FASTA/Q inputs. Among these works, RabbitFX [Zha+23] stands out as the state-of-the-art for parallel parsing of plain and gzipped FASTA/Q. However, since our work focuses on accelerating the parsing algorithm for a given thread, we will not evaluate RabbitFX in this chapter.

6.2. (Bitpacked) DNA representations

Depending on the application use case, different DNA representations might be desirable. Our approach produces three main representations: the traditional ASCII-based one, and two bitpacked versions (*packed* and *columnar*) in which each base is encoded with 2 bits. A convenient way to encode bases is to extract the second and third lowest bits from their ASCII representation, as described below:

A	C	T	G
0100**00**01	0100**01**11	0101**01**00	0100**11**11

However, sequences may still contain non-ACTG characters for IUPAC codes that we cannot bitpack losslessly, so we have to handle them separately. We propose two different ways to solve this. One solution is to treat non-ACTG characters as splits and only return contiguous chunks of ACTGs. Another solution is to use a lossy encoding that returns an additional bitmask marking ambiguous positions with ones (thus requiring 3 bits per base in total).

6.2.1. Packed format

Once each base is mapped to two bits, an intuitive way to represent the sequence is to pack each pair of bits consecutively, thus storing 4 bases in one byte. By convention, the first bases are mapped to the least significant bits and the bytes follow a little-endian order. This bitpacked representation is probably the most commonly used, and allows simple iteration on the bases by shifting and masking.

6.2.2. Columnar format

Another way to represent the sequence is to separate the stream of high bits and low bits. With this approach, 8 bases are stored in two bytes: one for the high bits and the other for the low bits. This representation maps each base of the sequence to exactly one bit in each part at the corresponding position, which is convenient for bitmask-based operations. For instance, locating `T`s in the sequence is as simple as intersecting the high bits and the negation of the low bits:

`seq`	`G`	`A`	`T`	`T`	`A`	`C`	`A`	`T`
`hi`	1	0	1	1	0	0	0	1
`lo`	1	0	0	0	0	1	0	0
`hi & !lo`	0	0	**1**	**1**	0	0	0	**1**

Again, note that the first bases on the left correspond to the least significant bits.

6.3. Streaming the input

Our approach follows three phases of *streaming*, *lexing* and *parsing*: the input is first streamed to a lexer that provides a tokenization of the stream, which is then fed to a parser to build the desired output.

The very first step of our algorithm is to stream the input that we want to parse. We support two main kinds of input: the ones that support random access (e.g. data in RAM or memory-mapped file) and the ones that have to be consumed sequentially (e.g. file reader or stdin). In particular, reader-based inputs allow transparent decompression of the data by detecting compressor-specific signatures ("magic bytes") at the start of the input. To handle all these types of input generically, we provide a unifying abstraction that streams the data by blocks of fixed size (in our case 64 bytes). While streaming these blocks is straightforward for data in memory and does not require any copy, we need a buffering strategy for reader-based input to amortize the cost of read syscalls.

The buffering strategy differs depending on the type of output produced. For bitpacked representations, the input buffer is only needed for tokenization so we simply use a fixed-size buffer (128 KB by default) and refill it when it is entirely consumed. For the ASCII-based representation, however, we want to reuse the sequence loaded in the buffer whenever possible and avoid copies. In that case, we use an adaptive strategy similar to Needletail [One19] or RabbitFX [Zha+23]. When a record is too large to fit in the current buffer, the size of the buffer is doubled. In practice, this occurs only for very long sequences such as chromosomes. When a record overlaps the boundary between two buffer fills, the partial record at the end of the current buffer is copied to the beginning of the next one before refilling.

6.4. Classifying the input with bitmasks

Unlike traditional lexing/parsing situations, we need to carefully design the structure of the lexing phase to be fully compatible with vectorized processing. Instead of consuming the input

and producing lexemes to be consumed later by the parser, the lexing phase will annotate the stream by producing bitmasks that encompass some information about the input. For instance, we will produce a bitmask that distinguishes parts of the input that are within the header of a FASTA record or within the sequence.

Producing bitmasks efficiently to classify the input is a classic way to produce highly efficient code. For instance, in simdjson [LL19], the lexing phase produces bitmasks for the position of structural characters in a JSON document. In rsonpath [GMP23], the small programs generating relevant bitmasks are denoted *classifiers*, and we will follow this terminology from now on.

Concretely, a bitmask gives a boolean information for each byte of the input. Those bytes are provided in streaming by blocks of 64 at a time. The goal of a highly performant lexer is to produce the desired bitmasks as efficiently as possible, and with the minimal number of branches. To illustrate this methodology, let us focus on the `header` bitmask, critical for the FASTA parsing in Helicase. This bitmask is true for a byte if and only if it belongs to a header of the document. Formally, it is delimited by the byte `>` and a line feed without any line feed in between, as described in Equation 6.1.

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
header	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0

To construct such a bitmask, it is essential to avoid scanning the input sequentially one byte at a time. The first step is to identify the relevant byte values in the input stream. Following the methodology used in `memchr`, we can efficiently produce bitmasks (with architecture-specific implementations discussed later) that mark the occurrences of the characters `>` and ↵.

The goal is then to derive a bitmask that identifies precisely the bytes located between a `>` and a ↵. Remarkably, this can be achieved without iterating over the bytes individually. Instead, the construction relies on carry propagation during the addition of two bit-vectors. This approach is efficient because modern processors perform additions on 64-bit integers with highly optimized carry propagation. In particular, the processor's carry flag enables fast propagation of carries across successive additions, allowing the required bitmask to be computed with only a few arithmetic and bitwise operations.

From automata to vectorized programs

It is possible, using the work of [PSS23; Soy23; Ser04], to analyze a given automaton and determine whether its execution can be expressed solely using bitwise bit-vector operations combined with addition. An automaton that admits an efficient program has several characterizations and belongs to the class of so-called *counter-free automata* [MP71]. Although these characterizations are decidable and conceptually well understood, there is currently no efficient implementation that automatically derives the corresponding vectorized program from the automaton. For small enough instances such as our lexer, the proof methodology of Paperman et al. [PSS23] can be applied manually to synthesize such a program.

We can produce a compact representation of the lexing phase through so called *vectorial circuits*. Those circuits can be thought of as formulas on bitmasks that can be evaluated on a given word. The results of the bitmask are the lexing annotations we need. Given a word u that we want to parse, we denote by $\mathbf{1}_{>}$ and $\mathbf{1}_{↵}$ the bitmask of positions where respectively the symbols `>` and ↵ occur. The formula to compute the `header` bitmask is then expressed as follows, using $+$ to denote an addition propagating the carry from left to right, $\neg$ to denote bitwise `not` and $\oplus$ to denote bitwise `xor`:

$$\texttt{header} := ((\mathbf{1}_{>} + \neg\mathbf{1}_{↵}) \oplus \neg\mathbf{1}_{↵}) \vee \mathbf{1}_{>}$$

The usage of carry propagation to encode part of the branching originates from Myers [Mye99]. Applying this formula to the previous example, we get the following:

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
$\mathbf{1}_{\text{↵}}$	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
$\mathbf{1}_{>}$	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
$\neg\mathbf{1}_{\text{↵}}$	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1
$\mathbf{1}_{>} + \neg\mathbf{1}_{\text{↵}}$	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1
$\oplus\neg\mathbf{1}_{\text{↵}}$	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0
$\vee\,\mathbf{1}_{>}$	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	0

6.5. Implementation of the lexing phase

> Technical section
>
> This section discusses technical details about architecture-specific assembly instructions.

While this formula provides us with the appropriate semantics, implementing it requires a bit more work since it is interpreted over a stream of 64-bytes blocks. We now describe its concrete implementation in the lexing phase as a *classifier*, i.e. an iterator that builds the annotation on-the-fly.

The first step of the lexing phase is to load the input block into a SIMD register, and to compute bitmasks for specific characters: `\n` for FASTQ, and both `\n` and `>` for FASTA. Additionally, we may want to detect non-ACTG bases to split the sequence or mark their positions. Both of these steps are described in Algorithm 6.1 and rely essentially on vectorized byte-wise comparisons followed by a `movemask`.

The `movemask` instruction, natively supported on x86 since SSE, extracts the most significant bit of each byte in a SIMD register into a scalar bitmask. However, this instruction is not available on ARM CPUs, so we have to implement it manually using multiple intrinsics, making it more expensive than on x86. A relatively efficient method to implement it on ARM is to make use of NEON's interleaved layout and byte-wise packing[1]. One benefit of having a manual implementation is that we can easily adapt it to extract multiple bits from each byte instead of a single one, which will turn out to be useful for the packed representation.

Algorithm 6.1: Pseudocode to detect newlines and non-ACTG characters.

function IsNewline(v_{data})
    $v_{\text{eq}} \leftarrow$ `cmp_eq`($v_{\text{data}}, v_{\text{↵}}$) *// test equality between each byte and ↵*
    return `movemask`(v_{eq}) *// return the corresponding bitmask (1 if equal, 0 otherwise)*

function IsACTG(v_{data})
    `LUT` $\leftarrow$ `["A", _, "C", _, "T", _, "G", _]` *// constant lookup table mapping bits to bases*
    $v_{\text{bits}} \leftarrow v_{\text{data}}$ `& 0b110` *// two bits encoding with a trailing zero*
    $v_{\text{lookup}} \leftarrow$ `lookup`(`LUT`, v_{bits}) *// map the bits to ACTG using the lookup table*
    $v_{\text{upper}} \leftarrow v_{\text{data}}$ `& !0b00100000` *// uppercase the input*
    $v_{\text{eq}} \leftarrow$ `cmp_eq`($v_{\text{lookup}}, v_{\text{upper}}$) *// test equality between uppercased input and lookup*
    return `movemask`(v_{eq})

[1] https://stackoverflow.com/questions/74722950

Depending on the type of desired output, we also use the lexing phase to produce a stream of columnar or packed sequences, both described in Algorithm 6.2. For the columnar representation, we simply shift each byte by 5 (resp. 6) positions to move the high (resp. low) bit to the beginning of the byte, which we can then extract using `movemask` as before. The packed representation, however, requires different strategies depending on the architecture. On x86, since `movemask` can only extract one bit at a time, we essentially reuse the high and low bits from the columnar format and interleave them. A simple way to interleave the bits is to use the `PDEP` instruction available with BMI2. However, this instruction is not supported by old CPUs, and has been notoriously slow on AMD CPUs until the Zen3 generation because it was microcoded. As a fallback for this kind of hardware, we also provide an alternative implementation that does the interleaving step on the *bytes* before the `movemask` rather than the bits. On the other hand, computing packed representations on ARM is much simpler since we can change the `movemask` implementation to extract two bits at a time, thus entirely avoiding the interleaving step. Due to these architectural differences, producing a packed representation is slightly more expensive than a columnar one on x86.

Algorithm 6.2: Pseudocode for the columnar and packed representations, on x86 and ARM.

function COLUMNAR(v_{data})
    $v_{\text{hi}} \leftarrow v_{\text{data}}$ `<<` `5` *// shift the third lowest bit to the beginning of the byte*
    $v_{\text{lo}} \leftarrow v_{\text{data}}$ `<<` `6` *// same for the second lowest bit*
    return `movemask`(v_{hi}), `movemask`(v_{lo}) *// movemask only extracts the highest bit of each byte*

function PACKEDX86(v_{data})
    $v_{\text{hi}} \leftarrow v_{\text{data}}$ `<<` `5`
    $v_{\text{lo}} \leftarrow v_{\text{data}}$ `<<` `6`
    $m_{\text{hi}} \leftarrow$ `movemask`(v_{hi})
    $m_{\text{lo}} \leftarrow$ `movemask`(v_{lo})
    return `interleave`($m_{\text{hi}}, m_{\text{lo}}$) *// interleave one bit of each mask, usually using `PDEP`*

function PACKEDARM(v_{data})
    $v_{\text{shift}} \leftarrow v_{\text{data}}$ `<<` `5`
    return $\texttt{movemask}_2(v_{\text{shift}})$ *// variant of movemask that extracts the highest two bits together*

6.6. Parsing relevant information with a control finite state machine

The parsing stage is implemented as a deterministic control layer operating over the output of the lexing classifiers. Rather than inspecting input bytes individually, the parser consumes bitmask summaries associated with fixed-size blocks, where each mask encodes the positions of structurally relevant symbols such as header markers and DNA characters. This design allows the parser to advance directly between meaningful positions using bitwise selection, avoiding per-character control flow.

The control logic for FASTA parsing is structured around a small number of states corresponding to the successive processing steps. Execution begins by locating the start of a header, after which the header line is consumed. The parser then enters an intermediate phase where it determines whether the subsequent region corresponds to another header or to DNA content. When a DNA region is detected, the parser initializes the necessary context and enters a tight processing loop over contiguous DNA symbols (restricted to `A/C/T/G` for some bitpacking strategies). This loop continues as long as the lexical masks indicate valid DNA positions. Upon encountering a boundary condition, such as a non-DNA symbol, a header marker, or the end of the input, the

current DNA segment is finalized and control returns to the intermediate phase. On FASTQ, the structure is similar but since each field spans over a single line, the corresponding states directly correspond to a given line count modulo 4, making it simpler overall.

All transitions are driven by predicates evaluated over the lexical masks. In practice, the parser repeatedly queries these masks to locate the next relevant position, using bitwise operations to identify the first occurrence of a header, DNA symbol or boundary. If no such position exists within the current block, the parser advances to the next block and repeats the process. As a result, large regions of unnecessary fields are skipped in constant time per block, and control decisions are made only at structurally significant locations.

The parser maintains a mutable state associated with the current record and DNA segment. Depending on the configuration, this state may include ranges for header and quality data, accumulators for DNA sequences, or alternative encodings such as packed and columnar representations. These data structures are updated incrementally during DNA processing and may be reset or extended when transitioning between segments. Boundary conditions determine when a DNA segment is considered complete, and user-defined policies control whether segments are split, merged or filtered.

The parsing stage can also produce events corresponding to record boundaries or DNA segments. These events are emitted at well-defined points in the control flow, typically when a new header is encountered or when a DNA segment is finalized. Event generation is entirely configuration-dependent and does not affect the structure of the control logic.

Overall, the parser follows a two-level organization in which a data-parallel lexing stage exposes structural information as bitmasks, and a compact control layer reacts to this information through a small set of states. This separation enables high-throughput streaming by eliminating fine-grained branching while preserving a simple and predictable execution model.

The parser's behavior is entirely dictated by a set of configuration parameters processed at compile time. These parameters control header processing, DNA accumulation strategy (string, packed, and/or columnar), boundary handling, and event emission. Conceptually, this design can be understood as generating a distinct parser implementation for each configuration, rather than a single generic parser containing runtime conditionals.

This approach corresponds to compile-time specialization (often referred to as "monomorphization"), where a generic implementation is transformed into multiple concrete variants. In contrast to designs relying on runtime branching (e.g. `if (compute_dna_columnar) ...`), each generated variant contains only the code relevant to the selected configuration, with all other paths removed during compilation.

6.7. Compile-time configuration and specialization

This specialization extends across both the parsing and lexing phases. Critical functions are aggressively inlined, and inlining propagates into the lexer itself. As a consequence, lexical computations that are not required under a given configuration are eliminated at compile time. For example, when DNA string accumulation is disabled, the corresponding logic for extracting and storing DNA bases is not generated, and the lexer omits the associated processing entirely. All the conditions governing buffer management, boundary semantics, and event emission are thus resolved statically. Each configuration hence produces a specialized instance of the control automaton in which transitions and associated actions are fixed in advance, with no residual dynamic branching. At the moment of writing, we provide 13 flags that can each trigger different behaviors. Altogether, this amounts to 8192 different possible specializations. More specialization could occur in the future to open the way for more efficient code.

6.8. Results

To evaluate the performance of our approach, we compared it against two state-of-the-art Rust libraries: Needletail 0.6.3 [One19] and Paraseq 0.4.8 [Tey25]. Needletail is the fastest single-threaded library to our knowledge, while Paraseq is getting increasingly popular because of its parallel-friendly design. The benchmarks presented in this section were run using the commit `5557b93` of our library. We measured the throughput of each parser in three main scenarios: parsing a genome in multiline FASTA format, short reads in FASTQ format and long reads in FASTQ format. The datasets used are detailed in Table C.1.

Table 6.1.: Description of the benchmarked datasets.

Dataset	Format	Size	# records	avg. seq. len.	line breaks	Link
Human genome (CHM13v2.0)	FASTA	3.16 GB	25	125 Mbp	every 80 bp	[2]
PacBio human HiFi reads	FASTQ	2.58 GB	92 K	14 Kbp	—	[3]
Illumina human short reads	FASTQ	2.23 GB	4 M	250 bp	—	[4]

The benchmarks are orchestrated through a bash script[5], which iterates over all input datasets, runs the parsers in release mode, and records the results in a CSV file. The compilation is configured to target the native microarchitecture of each machine, ensuring that the generated code fully exploits the available CPU features. Multiple parser configurations are evaluated to capture performance sensitivity to instruction-level differences. Each measurement is averaged over ten repetitions.

To ensure representative and robust measurements, the benchmarks were deployed across a wide range of heterogeneous machines available on Grid'5000 [Bal+13] (with different CPU generations and microarchitectures). Jobs were executed on reserved nodes in isolation from other workloads, ensuring that no external interference perturbs the results. This setup provides a broad and controlled view of performance across architectures. For completeness, the same benchmarks were also run on the author's personal laptop and desktop systems, allowing comparison with more conventional environments. A description of all the CPUs used across the benchmarks can be found in Table B.1 in appendix.

6.8.1. Parsing throughput

The main metric that we evaluated is the throughput of each parser on the different datasets. In order to have a fair comparison that reflects a typical execution, the performance was measured when reading the files from disk and are thus subject to more significant I/O bottlenecks than for data loaded in RAM. Figure 6.1 shows the throughput when parsing a human genome (in FASTA format) and short reads (in FASTQ format). An additional plot for long reads is available in Figure B.1 in appendix. For completeness, we also included results on data loaded in RAM in Figure B.2 and results comparing the throughput of collecting the sequence string and only counting the number of bases in Figure B.3 in appendix.

[2] https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0.fa.gz
[3] https://s3-us-west-2.amazonaws.com/human-pangenomics/NHGRI_UCSC_panel/HG002/hpp_HG002_NA24385_son_v1/PacBio_HiFi/15kb/m54328_180928_230446.Q20.fastq
[4] https://s3-us-west-2.amazonaws.com/human-pangenomics/NHGRI_UCSC_panel/HG002/hpp_HG002_NA24385_son_v1/ILMN/NIST_Illumina_2x250bps/D1_S1_L001_R2_007.fastq.gz
[5] https://github.com/imartayan/helicase/blob/main/bench/bench.sh

When parsing the human genome, Helicase is consistently twice as fast as its competitors on Intel CPUs, and roughly 50% faster on AMD and ARM CPUs. This trend is confirmed over all generations of CPUs, from the oldest ones only supporting SSE to those supporting AVX2 or NEON. We also observe that both bitpacking strategies (columnar and packed) are competitive with plain string parsing in this setting, which can be explained by the fact that the string version still has to eliminate the newlines contained in the sequence and thus cannot be copy-free. As expected given the extra interleaving step it requires, computing the packed representation is always slightly slower than the columnar one. Our implementation scales better with newer hardware. On old CPUs (e.g. Xeon X5670, 2010), the throughput is around 1.1 GB/s for Needletail and Paraseq and between 1.4 and 1.6 for variants of Helicase. Meanwhile, in the latest generation (Xeon Gold 6442Y, 2023), the throughput for Needletail and Paraseq ranges between 1.3 and 1.4 GB/s while Helicase achieves a throughput ranging from 3.7 to 4.7 GB/s. Hence, and contrary to the baseline, the implementation of Helicase clearly leverages hardware improvements of the last decade.

When parsing short reads, Helicase is consistently 10-30% faster than Needletail, itself faster than Paraseq. Compared to the results on the human genome, we observe that the overall throughput is higher and that the difference between Helicase and Needletail is less pronounced. This is mainly explained because FASTQ parsing requires less work (keeping track of the line count is sufficient to transition between states) and because the sequence of each record is not split across multiple lines, thus allowing a zero-copy strategy. On the other hand, bitpacked configurations do not benefit from this property and are comparatively slower than outputting plain strings. The results on long reads, shown in Figure B.1 in appendix, are even less pronounced. Having longer contiguous sequences makes the zero-copy strategy even more noticeable and file I/O becomes a bigger limitation.

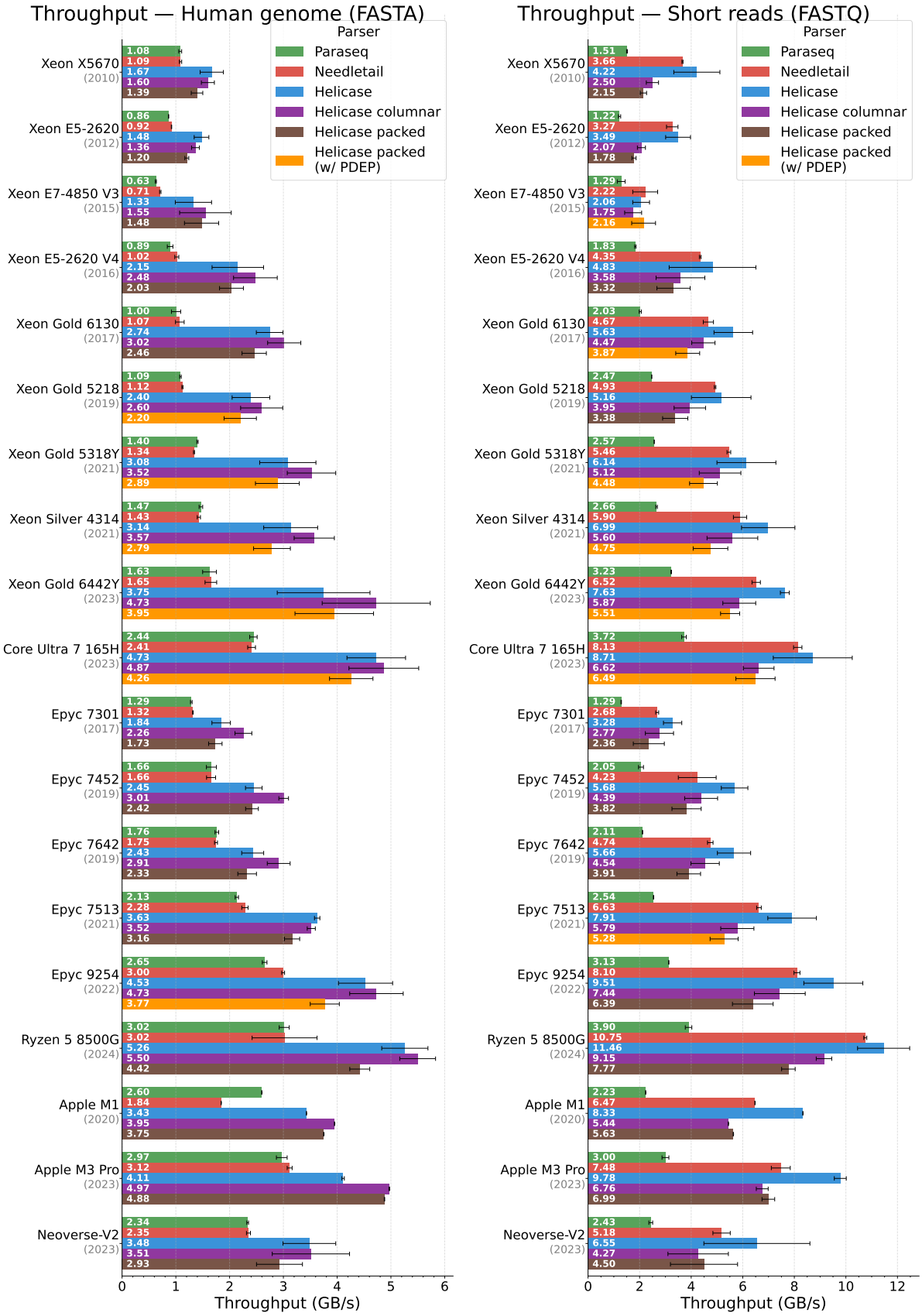
To illustrate the impact of compile-time specialization of the parser, we compared the throughput of collecting the sequence string for each record against a configuration that only counts the number of ACTG bases (skipping other IUPAC codes) on the human genome. This comparison, presented in Figure B.3 in appendix, confirms that a lighter configuration requirement successfully produces more efficient code.

6.8.2. Number of instructions, cycles and branches per byte

In addition to throughput, we measured the number of instructions, cycles and branches corresponding to each execution using perf counters, and divided it by the input size to have uniform values. Since perf counters are not as widely available on macOS, we did not include Apple CPUs for these metrics. Figure 6.2 shows the number of instructions and cycles per bytes for each parser on the human genome. Additional plots on short reads are available in Figure B.4 in appendix, as well as plots on the number of branches per byte in Figure B.5 and the number of branch misses per MB in Figure B.6.

These plots provide many interesting insights on the CPU characteristics and how well each parser exploits them. First of all, we note that the main factor impacting the number of instructions is the instruction-set supported by each CPU. This is especially noticeable when comparing Intel CPUs only supporting SSE (first two Xeon CPUs in Figure 6.2a, prior to 2013) to those supporting AVX2 afterwards: since SSE vectors are twice as small as AVX vectors, scanning a given block requires more instructions. When focusing on the packed representation, we can also observe the shift from microcoded `PDEP` instructions (first three Epyc CPUs, prior to 2020), which we avoided in favor of a fallback implementation shown in brown, to native `PDEP` instructions on AMD CPUs over the years.

Even though the number of instructions is roughly constant for a given instruction-set, the number of cycles spent to execute them is not. We observe in Figure 6.2b a decrease in the number of cycles over the years, reflecting the continuous improvements of CPU manufacturers. However, we can also notice that while Helicase has a number of instructions similar to Needletail,



(a) On a human genome (FASTA format).

(b) On short reads (FASTQ format).

Figure 6.1.: Throughput of each parser for FASTA and FASTQ files read from disk on multiple CPUs, sorted by manufacturer and year.

it uses significantly less cycles. This suggests that our implementation relies on instructions that the CPUs evaluate in parallel, making a better use of vectorized instructions, and therefore allows a good pipelining strategy. Finally, the results on the number of branches (shown in Figure B.5 in appendix) confirm that Helicase successfully eliminates more branches with our vectorization strategy.

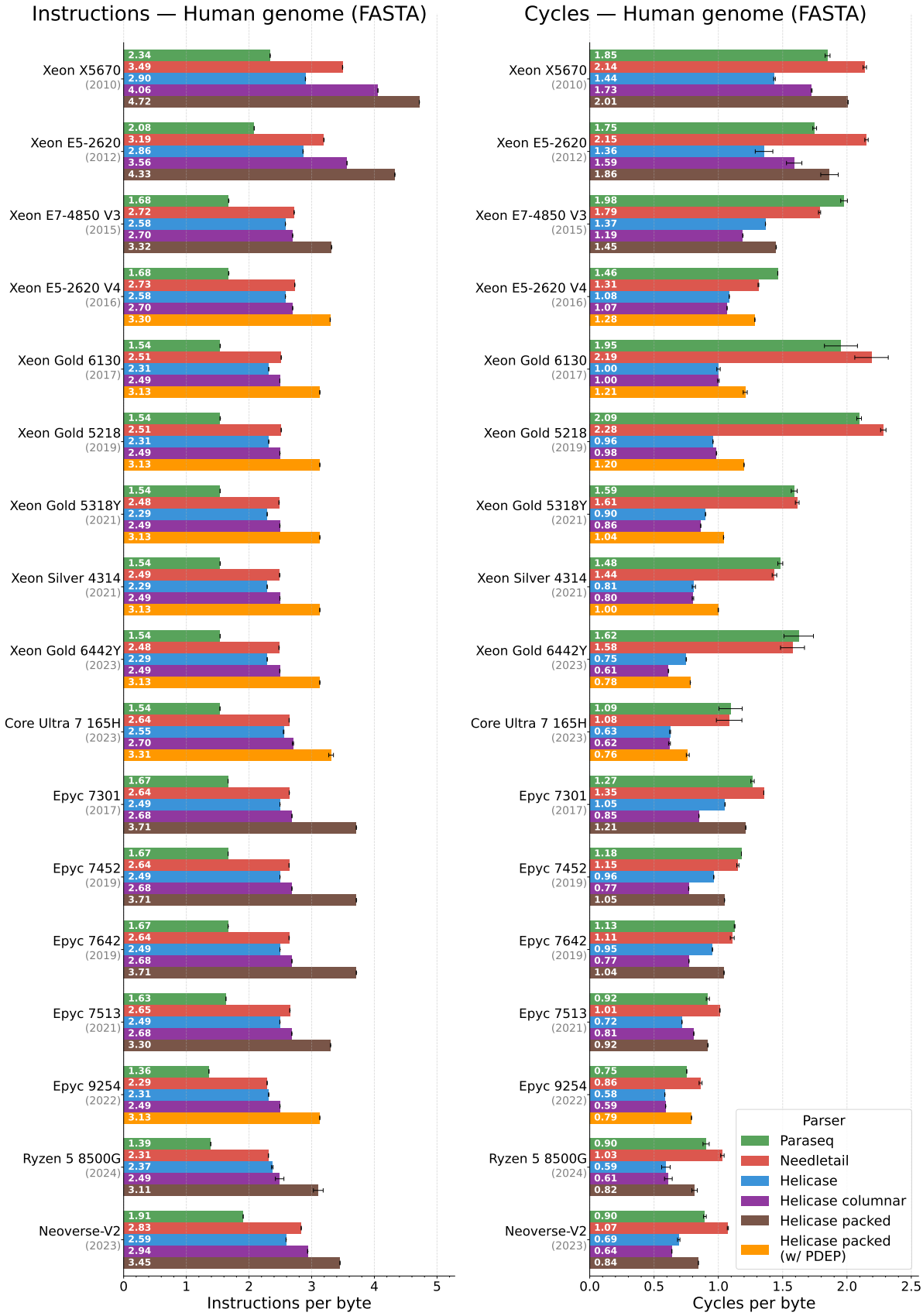
6.9. Conclusion

We presented Helicase, a high-throughput Rust library for parsing FASTA and FASTQ files that exploits SIMD vectorization to maximize single-threaded throughput on both x86 and ARM. At the core of our approach is a vectorized lexing stage based on bitmask classifiers derived from the theory of counter-free automata [PSS23; Ser04]. Rather than scanning the input byte by byte, the lexer annotates 64-byte blocks in parallel using carry-propagating arithmetic and bitwise operations, fusing the detection of all structural markers into a single pass. The parser is then structured as a finite state machine over these bitmasks, and specialized at compile time to only contain the code paths relevant to the requested fields. Helicase is able to produce on-the-fly DNA bitpacking for two compact representations: a *packed* format and a *columnar* format that separates the high and low bits, with non-ACTG characters handled either as segment boundaries or via a lossy encoding with an additional ambiguity mask. We also interfaced Helicase with the existing libraries developed for SimdMinimizers [GM25], to support efficient k-mer hashing and minimizer computation for the packed format. In particular, we will discuss in the next chapter how packed representations can be used to accelerate rolling hashes.

We evaluated Helicase on a wide collection of CPUs, deployed reproducibly on Grid'5000. On FASTA genome parsing, Helicase is up to 2× faster than Needletail on Intel CPUs and 50% faster on AMD and ARM, while matching or exceeding it on all FASTQ workloads. For data loaded in RAM, Helicase reaches the core memory bandwidth (49 GB/s on an Apple M3 Pro for long reads), confirming that the vectorized design successfully eliminates the parsing overhead as a bottleneck.

A number of avenues remain to extend and improve our approach. On the memory management side, DNA records are currently stored entirely in main memory, which is sufficient for most organisms but may fall short for certain species whose chromosome sizes exceed human genome standards by an order of magnitude [Sch+24]. As such large genomes accumulate through ambitious sequencing programs [Pen17], introducing a configurable buffer size to process DNA records under memory constraints is a natural next step. On the hardware side, implementing a version optimized for AVX-512 could provide substantial performance gains on latest CPUs. Parallelization is another important direction: since Helicase processes a file as a single stream, one could instead partition the input into multiple chunks and assign each to a separate reader, avoiding the synchronization overhead of a producer-consumer model. On the algorithmic side, there is potential to further exploit the columnar DNA representation by designing specialized algorithms for k-mer generation and hashing or pattern matching.

In addition to FASTA and FASTQ, Helicase could also be extended to parse annotations in GFF and GTF formats efficiently. Another extension involves packaging Helicase for interpreted languages such as Python and JavaScript. While this would broaden accessibility, it introduces challenges related to managing the configuration state and controlling runtime code size. Finally, building additional tooling on top of the core library, such as utilities for format conversion, filtering or statistics, would further integrate Helicase into everyday workflows.



(a) Instructions per byte on a human genome.

(b) Cycles per byte on a human genome.

Figure 6.2.: Instructions and cycles per byte on multiple CPUs, sorted by manufacturer and year.

7. Rolling hashes on sequences

> **Note**
>
> This chapter is partially adapted from Groot Koerkamp and Martayan (2025), accepted to SEA 2025 and published in LIPIcs.
> Ragnar Groot Koerkamp designed the initial AVX2 implementation, which we then improved and turned into the `packed-seq` and `seq-hash` libraries together. The paper itself has equal contributions from both of us. The analysis of NtHash biases is my own later work, and is not part of the original publication.

We've seen in Chapter 6 how to parse and bitpack sequences efficiently. Hashing their content, in particular the k-mers they contain, is one of the most common operations in sequence analysis. In this chapter, we will focus on rolling hashes, which are especially well-suited for this task, and see how far we can optimize them.

7.1. Rolling hashes and their properties

The concept of *rolling hash* (or *recursive hash*) was first introduced by Karp and Rabin [KR87] for pattern matching. Its purpose is to efficiently compute a hash over a sliding window of integers, without having to rehash the content of the entire window as it shifts.

Karp and Rabin [KR87] defined their rolling hash as follows: given a large prime number p and $r \in [2, p)$,

$$H(x_0 \dots x_{k-1}) = \sum_{i=0}^{k-1} r^{k-1-i} x_i \mod p$$

7.1.1. Constant-time updates

The most important property of rolling hashes is that removing the first value of the window and adding a new value at the end only requires a constant-time update to the hash.

To remove x_0, we can simply subtract the corresponding term:

$$H(x_1 \dots x_{k-1}) = H(x_0 \dots x_{k-1}) - r^{k-1} x_0 \mod p$$

Note that, assuming the window size k is known in advance, r^{k-1} can be precomputed once. Similarly, to append a new value x_k, we can shift the previous values by one position to the left by multiplying by r before adding the new term:

$$H(x_1 \dots x_k) = H(x_1 \dots x_{k-1}) \cdot r + x_k \mod p$$

By combining these two operations, we can compute the hashes of sliding windows in a text T in $\mathcal{O}(|T|)$ regardless of the window size. In the context of bioinformatics, this is especially useful for hashing k-mers, including large ones that would not fit in a native integer.

7.1.2. Merging and splitting hashes

Another useful property is that the hashes of two sequences u and v can be merged in constant time:

$$H(uv) = H(u) \cdot r^{|v|} + H(v)$$

And the same relation can be used to remove any prefix of a given hash:

$$H(v) = H(uv) - H(u) \cdot r^{|v|}$$

In particular, the merging property can be used to parallelize the hash computation of a large sequence by dividing it into chunks, hashing them in parallel and merging the resulting hashes.

7.1.3. Hashing arbitrary substrings using prefixes

While rolling hashes are traditionally used to hash fixed-sized windows, we can also use them to compute the hashes of arbitrary substrings.

One way to achieve this is to compute the hashes of every prefix of the text T and then use the splitting property to subtract the corresponding prefixes:

$$H(T[i, j)) = H(T[0, j)) - H(T[0, i)) \cdot r^{j-i}$$

This technique allows any substring of T to be hashed in constant time, at the cost of preindexing the prefix hashes in $\mathcal{O}(|T|)$ time and space.

7.2. NtHash and its variants

7.2.1. Cyclic polynomial hashing

One of the drawbacks of Rabin-Karp hashing is that it requires multiplying and computing a modulo for every update, which is quite expensive compared to other operations[1]. Cohen [Coh97] proposed an alternative rolling hash, based on cyclic polynomials, that replaces the multiplication by a bitwise left rotation (denoted rol) and the addition by an exclusive or (denoted $\oplus$):

$$H(x_0 \dots x_{k-1}) = \bigoplus_{i=0}^{k-1} \mathrm{rol}^{k-1-i}(h(x_i))$$

Note that each x_i is replaced by a seed value $h(x_i)$, typically computed using a lookup table, to have a better distribution of the bits.

Using this definition, we have

$$H(x_1 \dots x_{k-1}) = H(x_0 \dots x_{k-1}) \oplus \mathrm{rol}^{k-1}(h(x_0))$$

and

$$H(x_1 \dots x_k) = \mathrm{rol}(H(x_1 \dots x_{k-1})) \oplus h(x_k)$$

Again, assuming the window size k is known in advance, $\mathrm{rol}^{k-1}(h(\cdot))$ can be precomputed and stored in a separate lookup table.

[1] Computing a modulo requires doing a division which is typically an order of magnitude slower than bitwise operations.

7.2.2. NtHash and MulHash

NtHash [MCVB16] is a direct adaptation of cyclic polynomial hashing, specialized for the DNA alphabet `A/C/T/G`. The main benefit is that since the lookup table only needs to store 4 hashes, it can fit in a 128-bit (resp. 256-bit) register for 32-bit (resp. 64-bit) hashes, making it convenient for vectorization. A scalar implementation of ntHash is detailed in Algorithm 7.1.

Another variation, introduced as MulHash in [GM25], is to replace the lookup table by a simple multiplication: $h(x_i) = C \cdot x_i$ where C is a fixed random constant. This approach is slightly slower than a lookup table for small alphabets, but faster for larger alphabets such as ASCII inputs.

Algorithm 7.1: Scalar pseudocode for ntHash.

function NTHASH(k, `in_out`) *// `in_out` iterates over (last bp, first bp) of each k-mer*
- $T_{\text{in}} \leftarrow [h(\texttt{A}), h(\texttt{C}), h(\texttt{T}), h(\texttt{G})]$ *// static lookup tables*
- $T_{\text{out}} \leftarrow [\text{rol}^{k-1}(h(\texttt{A})), \text{rol}^{k-1}(h(\texttt{C})), \text{rol}^{k-1}(h(\texttt{T})), \text{rol}^{k-1}(h(\texttt{G}))]$
- $H \leftarrow 0$
- **for** (`in_bp`, `_`) in `in_out`$[0 : k-1]$ **do** *// no `out_bp` in the first $k-1$ iterations*
  - $H \leftarrow \text{rol}(H)$ *// implemented as $H \leftarrow (H \ll 1 \,|\, H \gg 31)$*
  - $H \leftarrow H \oplus$ `table_lookup`$(T_{\text{in}}$, `in_bp`$)$
- **for** (`in_bp`, `out_bp`) in `in_out`$[k-1 :]$ **do** *// 2-bit bp coming in and out of each k-mer*
  - $H \leftarrow \text{rol}(H)$
  - $H \leftarrow H \oplus$ `table_lookup`$(T_{\text{in}}$, `in_bp`$)$
  - **yield** H
  - $H \leftarrow H \oplus$ `table_lookup`$(T_{\text{out}}$, `out_bp`$)$

7.2.3. Reverse-complement and canonical hashing

This definition of rolling hashes is also convenient to hash the reverse-complement of a sequence. Since we process the values in a reverse order, we can simply change the direction of rotation:

$$\overline{H}(x_0 \dots x_{k-1}) = \bigoplus_{i=0}^{k-1} \text{rol}^i(h(\overline{x_i}))$$

In that case, we have

$$\overline{H}(x_1 \dots x_{k-1}) = \text{ror}\left(\overline{H}(x_0 \dots x_{k-1}) \oplus h(\overline{x_0})\right)$$

and

$$\overline{H}(x_1 \dots x_k) = \overline{H}(x_1 \dots x_{k-1}) \oplus \text{rol}^{k-1}(h(\overline{x_k}))$$

where $\overline{x_i}$ denotes the complement of x_i and ror denotes a bitwise right rotation.

Since we can easily maintain the hash of both a sequence and its reverse-complement, we can add both hash values to obtain a canonical hash: $\widehat{H} = H + \overline{H}$. Note that any symmetric operation could work here, but something like a min would introduce a bias in the high bits.

7.3. Vectorized implementation of ntHash

Because it only relies on bitwise operations and has no branches, Algorithm 7.1 is a good candidate for vectorization. In order to make full use of the 256-bit registers available on most x86 CPUs, our goal in this section will be to compute 8 32-bit hashes at the same time. The main limitation here is that rolling hashes are inherently sequential: each hash value is updated from the hash of the previous window. Therefore, instead of computing 8 consecutive hashes, we focus on computing 8 *independent* hashes together.

7.3.1. Parallel iteration on a packed input

As input to our algorithm, we use the 2-bit *packed* sequence representation as described in Section 6.2. Since we want to compute 8 independent hashes, we first split the sequence into 8 chunks and assign each chunk to a dedicated *lane* of a SIMD register. Note that the chunks overlap by $k-1$ bases so that each window of size k is processed once. In order to have the content of each chunk available in separate lanes, we load one 256-bit register per chunk and perform a *transpose* operation[2]. Once each chunk is loaded in a separate lane, we can iterate over the bases of each chunk simultaneously by shifting and masking the bottom 2 bits of each lane. The entire iteration pipeline is illustrated in Figure 7.1.

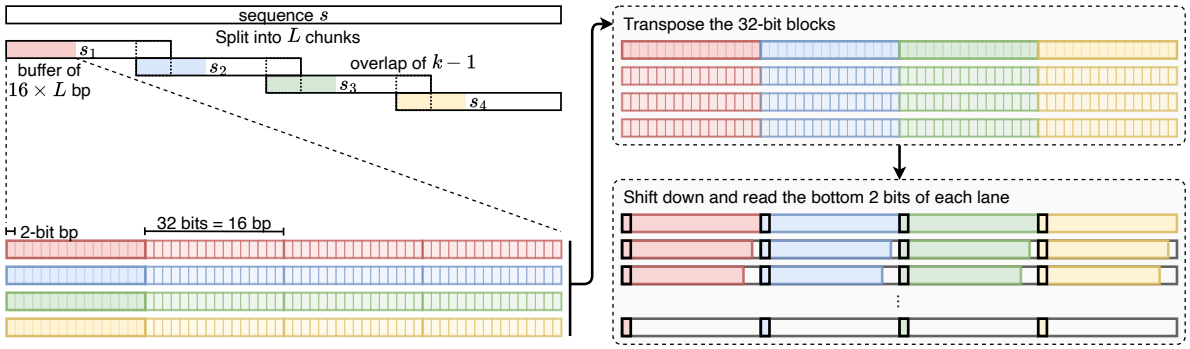


Figure 7.1.: Overview of parallel iteration on a packed sequence, each color corresponds to a SIMD lane associated to a given chunk. Note that while the figure shows $L = 4$ lanes, we use $L = 8$ lanes in practice for 256-bit registers.

This routine for parallel iteration over packed sequences is available in a Rust library named packed-seq.

7.3.2. Vectorized operations and table lookups

Using this approach for parallel iteration, we produce two iterators for the bases coming in and out of each k-mer. We can then reuse Algorithm 7.1 and replace each scalar operation with the corresponding vectorized operation on 8 lanes. The main difference is for the `table_lookup` that computes $h(x_i)$ or $\text{rol}^{k-1}(h(x_i))$ for each lane, which we implemented using `_mm256_permutevar_ps` in AVX2 and `vqtbl1q_u8` in NEON. Our vectorized implementation of ntHash and MulHash, built on top of packed-seq, is available in the seq-hash Rust library.

7.3.3. Performance

To evaluate the performance gain brought by vectorization, we measured the throughput of different hashing methods on x86 and ARM. We benchmarked 5 hashing methods in Rust (script available at https://github.com/imartayan/nthash-bias/blob/main/src/bin/throughput.rs):

- iterating over ASCII-encoded k-mers (slice of 31 bytes) and hashing them with `rapidhash`[3] v4.4.1 (which is one of the fastest hashes for slices of bytes)
- iterating over bitpacked k-mers (stored in a `u64`) with `packed-seq` and hashing them with `fxhash`[4] v2.1.2 (which is one of the fastest hashes for native integers)
- ntHash implementation from the `nthash`[5] crate v0.5.1
- our implementation of ntHash (`seq-hash` v0.1.2)

[2] https://stackoverflow.com/questions/25622745
[3] https://docs.rs/rapidhash/latest/rapidhash/
[4] https://docs.rs/rustc-hash/latest/rustc_hash/
[5] https://docs.rs/nthash/latest/nthash/, not to be confused with `nthash-rs` which is much slower

- our implementation of mulHash (`seq-hash` v0.1.2)

The results are reported in Table 7.1. Note that enabling the most aggressive compiler optimizations (`lto="fat"` and `codegen-units=1`) has a big impact on the throughput, especially for the `nthash` crate.

Table 7.1.: Throughput of different hashing methods on x86 and ARM using a single thread, best in bold.

Implementation	Throughput on Xeon Gold 6430 (x86)	Throughput on Apple M1 (ARM)
ASCII seq + `rapidhash`	0.47 Gbp/s	0.65 Gbp/s
`packed-seq` + `fxhash`	0.71 Gbp/s	1.25 Gbp/s
`nthash` crate	1.03 Gbp/s	1.00 Gbp/s
`seq-hash` ntHash	**2.65 Gbp/s**	2.28 Gbp/s
`seq-hash` MulHash	2.39 Gbp/s	**2.30 Gbp/s**

Overall, our vectorized implementation is 2.3–2.6× faster than the existing ntHash implementation, and is the fastest option among evaluated methods. We can also notice that using a packed input also benefits non-rolling hashes since the input is 4× smaller.

7.4. Breaking ntHash

While ntHash is clearly not meant to be a cryptographic hash, it still has some flaws that can also happen in non-adversarial use cases.

7.4.1. Forging and propagating collisions

The first observation we can make is that it's very easy to create collisions when the window size k is larger than the number of bits in the hash (which is not so surprising because the function cannot be injective in this case). Assuming we are still using 32-bit hashes, any pair of bases that are 32 (or any multiple of 32) positions apart will end up with the same rotation (since rotating by 32 is equivalent to not rotating at all). Therefore, for any $\alpha, \beta \in \Sigma$ and $u \in \Sigma^{31}$,

$$H(\alpha \cdot u \cdot \beta) = H(\beta \cdot u \cdot \alpha)$$

and similarly

$$H(\alpha \cdot u \cdot \alpha) = H(\beta \cdot u \cdot \beta) = \mathrm{rol}(H(u))$$

This issue is discussed in ntHash2 [Kaz+22], where the authors suggest to replace the rol operation with srol, which divides the 64-bit hash into a 31-bit and a 33-bit part and rotates them independently. While this operation is more expensive to compute, it has a period of 1023 instead of 32, thus limiting these specific collisions to $k \geq 1024$.

Something more problematic is that, because of the merging property (described in Section 7.1.2), if for some word size n we have a collision for $u \neq v \in \Sigma^n$, it will create Σ^{k-n} collisions for each word size $k \geq n$:

$$\forall u, v \in \Sigma^n, \forall x, y \in \Sigma^*, H(u) = H(v) \implies H(x \cdot u \cdot y) = H(x \cdot v \cdot y)$$

Therefore, it is especially important to avoid collisions for small words. Ragnar Groot Koerkamp described on his website[6] a method to select the seed values $h(x_i)$ to avoid collisions in 64-bit hashes for $k \leq 32$. His solution is essentially to define $h(\mathtt{T})$ as $h(\mathtt{A}) \oplus h(\mathtt{C}) \oplus h(\mathtt{G})$ and to adjust the other three values to obtain an invertible system.

[6] https://curiouscoding.nl/posts/nthash/

7.4.2. Bias on the leading zeros

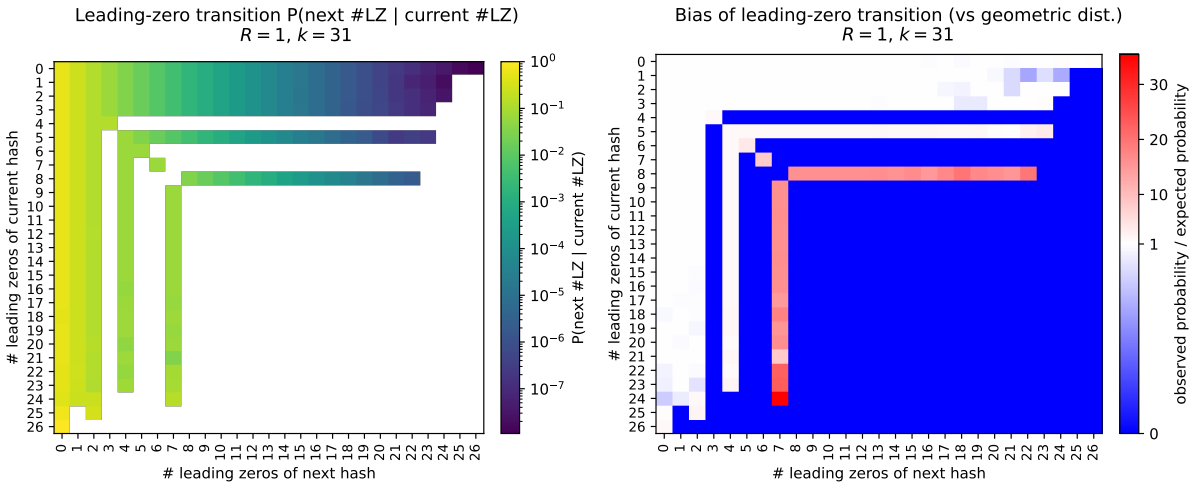
A very common use case of rolling hashes in bioinformatics is to hash every k-mer and select those with "small" hashes as representatives. For instance, this is extensively used in MinHash-based sketching techniques (presented in Chapter 3) or in random minimizers (presented in Chapter 4), which we will discuss in more detail in the next chapter.

These techniques generally assume that the hash values are independent and uniformly distributed. Under these assumptions, every k-mer has a probability 2^{-n} that its hash starts with at least n zeros. So if we define "small" as starting with n zeros, the distance between "small" k-mers roughly follows a geometric distribution (this is not exactly true because of repeated k-mers, though this is negligible for a large k).

Unfortunately, I noticed that the empirical distribution I had in my experiments[7] was not quite geometric, and that the repartition of "small" k-mers was skewed. I observed in particular that the number of leading zeros for a given k-mer was very correlated with the number of leading zeros of the neighboring k-mers, thus invalidating both independence and uniformity assumptions.

To document this behavior, I computed the number of leading zeros for every k-mer hash in a large random sequence (4 billion bases), and measured the probability that two consecutive k-mers transition from i leading zeros to j leading zeros. These experiment scripts are available at https://github.com/imartayan/nthash-bias.

Figure 7.2 contains two visualizations of this transition probability. The first one (Figure 7.2a) shows the raw probability of each transition: each line corresponds to the current number of leading zeros, while each column corresponds to the next value. The second one (Figure 7.2b) shows the ratio between this probability and the expected probability for a geometric distribution: blue when less than expected, white when matching the expectation and red when more than expected.



(a) Leading-zero transition for $R = 1$.

(b) Bias relative to a geometric distribution for $R = 1$.

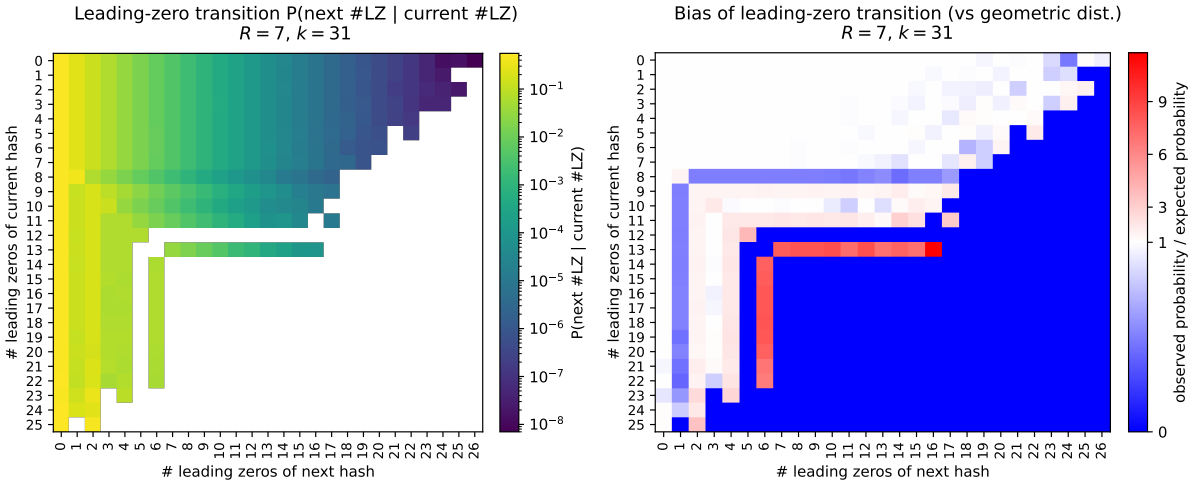
Figure 7.2.: Leading-zero bias with $R = 1$ on a random sequence using $k = 31$.

The first thing that we can notice in Figure 7.2a is that many of the cells are actually empty, meaning that the corresponding transition is simply not observed. While this is not very surprising for the bottom-right half of the matrix simply because of sample size (remember that a transition from i to j has a probability proportional to $2^{-(i+j)}$ in theory), it is much more surprising to see empty rows or columns interleaved with non-empty ones (see columns 3, 5, 6

[7] documented at https://github.com/imartayan/pfp-distribution

and rows 4, 6, 7). In particular, this implies that after observing a hash with exactly 4 leading zeros, the hash next to it will always have at most 3 leading zeros. Conversely, a hash with exactly 3 leading zeros can only be observed after at most 4 leading zeros. Figure 7.2b allows us to put these probabilities into perspective with the expected ones. While the first rows and columns mostly match the expectations, the missing transitions in blue are compensated by overrepresented ones in red. Notably, after observing a hash with at least 9 leading zeros, the probability of having exactly 7 leading zeros next to it is 16× higher. This creates an interesting dynamic where small hashes with at least 9 leading zeros are likely followed by 7 leading zeros, which are then followed by at most 6 leading zeros.

Even though this phenomenon is difficult to explain, the offset of one between the missing rows and columns made me conjecture that it is related to the 1-bit rotations performed at each step. To verify this hypothesis, I introduced variations of ntHash where the 1-bit rotations are replaced by R-bit rotations. This does not cause any problem as long as R is odd (since R and 32 need to be coprime) and does not affect performance, but simply shuffles the rotation offsets at each step. Figure 7.3 shows the results obtained for $R = 7$ while Figure 7.4 shows the results for $R = 13$.



(a) Leading-zero transition for $R = 7$.

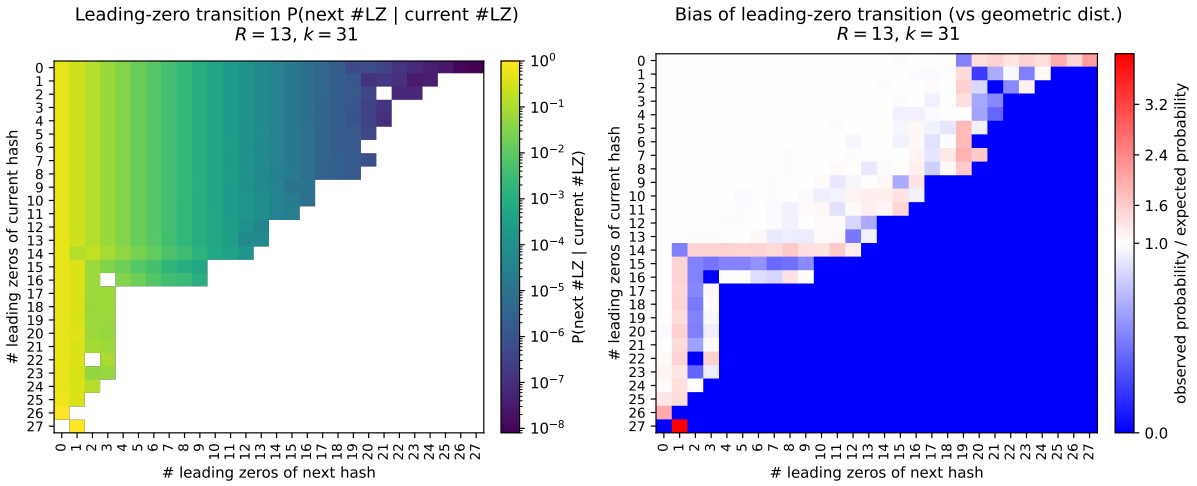
(b) Bias relative to a geometric distribution for $R = 7$.

Figure 7.3.: Leading-zero bias with $R = 7$ on a random sequence using $k = 31$.

Interestingly, R does have an impact on the offset between biased rows and columns: in Figure 7.3b we observe a pattern starting at $(i, j) = (8, 1)$ while in Figure 7.4b we observe two patterns starting at $(i, j) = (14, 1)$ and $(i, j) = (0, 19)$ respectively (note that $19 = -13 \mod 32$), which confirms the $i - j = R \mod 32$ conjecture. One practical benefit of shifting the biased rows and columns is that their probability drastically drops, making them less concerning in practice and leading to a more regular transition matrix, as illustrated in Figure 7.3a and Figure 7.4a. As a result, I recommend using $R = 13$ or 15 by default to reduce the bias (and we implemented a similar fix in `seq-hash`).

7.5. Toward minimizer computation

The vectorized hashing routines developed in this chapter feed directly into Chapter 8, where we use the resulting stream of k-mer hashes to extract minimizers efficiently.



(a) Leading-zero transition for $R = 13$.

(b) Bias relative to a geometric distribution for $R = 13$.

Figure 7.4.: Leading-zero bias with $R = 13$ on a random sequence using $k = 31$.

8. Vectorized computation of minimizers

> **Note**
>
> This chapter is adapted from Groot Koerkamp and Martayan (2025), accepted to SEA 2025 and published in LIPIcs.
> Ragnar Groot Koerkamp designed the initial AVX2 implementation, which we then improved and turned into the `simd-minimizers` library together. The paper itself has equal contributions from both of us.

Chapter 7 covered efficient implementations of rolling hashes, allowing us to hash k-mers at high throughput. These k-mer hashes are then typically used to partition or select representative k-mers. One major such application is computing *random minimizers* (introduced in Chapter 4): for each window of w consecutive k-mers, we select the smallest k-mer given a pseudo-random order defined by hash values.

In this chapter, we focus on computing random minimizers as efficiently as possible. More precisely, we are interested in finding the *starting positions* of random minimizers in a sequence, from which we can extract the minimizers themselves. We will first focus on computing *forward* minimizers (without taking reverse-complement into account) for simplicity, and later describe a solution for *canonical* minimizers in Section 8.3. Our implementation of the vectorized algorithm discussed in Section 8.4 is available in the simd-minimizers Rust library.

The computation of random minimizers is divided in three main steps, illustrated in Figure 8.1. First each k-mer of the input string (corresponding to horizontal lines in the figure) is hashed to a 32-bit integer (value above each line) using a rolling hash as presented in Chapter 7. Then, for each window of w consecutive k-mers, we find the absolute position of its smallest hash: for instance the fourth window in the figure, highlighted in yellow, has a smallest hash equal to 12 starting at position 5 (zero-based). Finally, the resulting minimizer positions are deduplicated.

8.1. Existing algorithms for sliding window minimum

Assuming that we have computed the hash values, our problem becomes a *sliding window minimum*. Given a sliding window of w values (in our case k-mer hashes), we want to output the position of its smallest value (breaking ties with the leftmost one). Many approaches can be used to solve this problem, here we will focus on three common ones: the "naive" algorithm, the "rescan" algorithm and the monotone queue algorithm. A summarized comparison of these three algorithms is given in Table 8.1.

8.1.1. Naive approach

The simplest approach is to simply loop over the w values in each window independently, as described in Algorithm 8.1. This takes $\mathcal{O}(nw)$ time, where n denotes the total number of values, but has the benefit of being branchless and can still be quite efficient when w is small by using vectorized instructions.

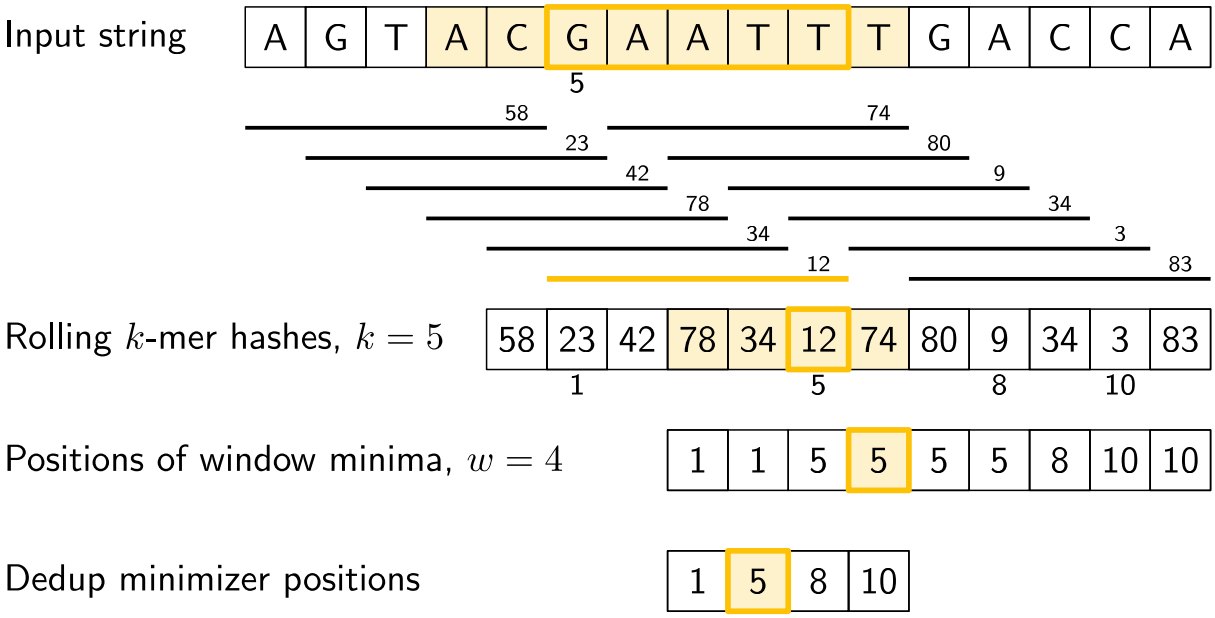


Figure 8.1.: Overview of the main steps to compute random minimizers, for $k = 5$ and $w = 4$.

Algorithm 8.1: Naive algorithm for sliding window minimum.

function NAIVE(w, `vals`)
  for i in $\{0, \dots, |\texttt{vals}| - w\}$ **do**
    yield $\arg\min\{\texttt{vals}[i], \dots, \texttt{vals}[i + w - 1]\}$

8.1.2. Monotone queue

A solution with better complexity, detailed in Algorithm 8.2, is to use a *monotone queue*, which stores a non-decreasing subsequence of the w hashes, alongside their positions. Every time the window slides one to the right and we are about to push a new k-mer hash onto the right of the queue, we first remove any values larger than it, as they are "shadowed" by the new hash and can never be minimal anymore. The minimum of the window is then always the leftmost queue element. This data structure guarantees an amortized constant time update, but has many unpredictable branches due to removing between 0 and w values, which makes it costly in practice. Because of this constant-time guarantee, monotone-queue based approaches have been adopted by tools such as Bifrost [HM20].

Algorithm 8.2: Monotone queue algorithm for sliding window minimum.

function MONOTONEQUEUE(w, `vals`)
  `queue` $\leftarrow$ empty DOUBLEENDEDQUEUE
  for i in $\{0, \dots, |\texttt{vals}| - 1\}$ **do**
    if `queue.front().pos` $+ w \leq i$ **then**
      `queue.pop_front()`
    while `queue.back.val` $>$ `vals`$[i]$ **do**
      `queue.pop_back()`
    `queue.push_back(val` $\leftarrow$ `vals`$[i]$, `pos` $\leftarrow i)$
    if $i \geq w - 1$ **then**
      yield `queue.front().pos`

8.1.3. Rescan

Another approach used in bioinformatics is to only keep track of the minimum value and rescan the entire window of w values when the current minimum goes out of scope, as described in Algorithm 8.3. This method was notably popularized by minimap2 [Li18]. While this algorithm does not guarantee a worst-case constant time update, it only branches when the minimum goes out of scope and hence is more predictable. This makes it more efficient in practice, especially since minimizers typically have a density of $\mathcal{O}(1/w)$ so that the $\mathcal{O}(w)$ rescan step takes amortized constant time per element.

Algorithm 8.3: Rescan algorithm for sliding window minimum.

```
function RESCAN(w, vals)
    min_val ← +∞
    min_pos ← 0
    for i in {0, ..., |vals| − 1} do
        if vals[i] < min_val then
            min_val ← vals[i]
            min_pos ← i
        if min_pos + w ≤ i then
            min_pos ← arg min{vals[i − w + 1], ..., vals[i]}
            min_val ← vals[min_pos]
        if i ≥ w − 1 then
            yield min_pos
```

Table 8.1.: Comparison of the naive, rescan and monotone queue sliding window minimum algorithms.

Algorithm	Time complexity	Branches
Naive	$\mathcal{O}(nw)$ worst-case	branchless
Rescan	$\mathcal{O}(nw)$ worst-case, $\mathcal{O}(n)$ on average	branch for each rescan
Monotone queue	$\mathcal{O}(n)$ worst-case	branch for each update

8.2. A branchless linear algorithm using two stacks

8.2.1. The two-stacks algorithm

Ideally, we would like to design an algorithm that has both a $\mathcal{O}(n)$ worst-case complexity (like the monotone queue) and no unpredictable branches (like the naive approach). The *two-stacks* method [HST17; TKPP18], well-known in the competitive programming community[1], allows us to compute *online* sliding minima where elements may be added and removed at varying rates. Here, we only discuss a version where the number of elements remains constant at w.

Conceptually, we first split the sequence of input k-mer hashes into blocks of size w, as shown in Figure 8.2. Then, we can compute both prefix minima and suffix minima of each block in $\mathcal{O}(w)$ per block, or $\mathcal{O}(1)$ amortized per input hash. Now, any window of size w can be split into a suffix of the previous block and a prefix of the current block (as highlighted in yellow in Figure 8.2), and we can return the minimum of the two corresponding suffix/prefix minima. When the window exactly coincides with a block (as shown on the right), the suffix and prefix minimum are equal.

[1] https://codeforces.com/blog/entry/71687

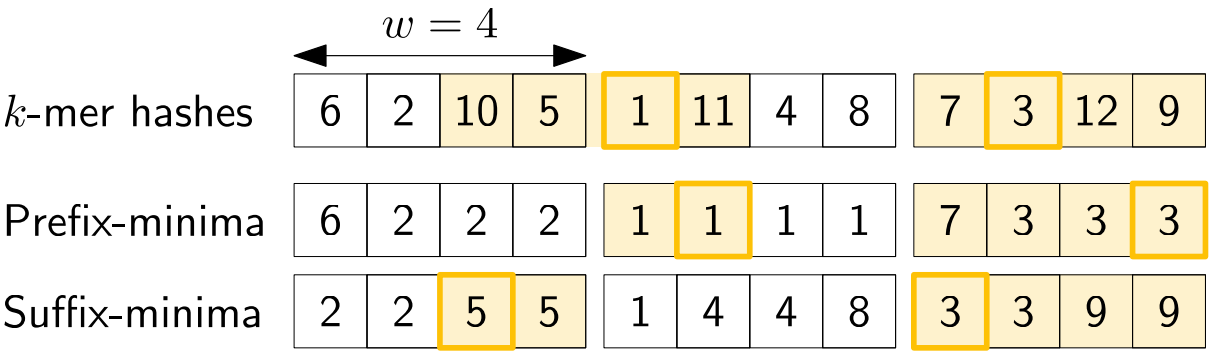


Figure 8.2.: An example showing how sliding-window minima can be computed for windows of size $w = 4$. The first window highlighted in yellow overlaps two blocks and has 5 as suffix minimum and 1 as prefix minimum. The second highlighted window coincides with a block and has 3 as suffix and prefix minimum.

In the implementation, detailed in Algorithm 8.4, the prefix minima are simply computed incrementally, while the suffix-minima are computed in batches after every block of w hashes has been filled. This way, a single buffer of size w is needed and the two cases above are unified. The only branch in the algorithm triggers exactly every w iterations, and is thus completely data-independent and predictable.

Algorithm 8.4: Two-stacks algorithm for sliding window minimum.

```
function TwoStacks(w, vals)
    prefix_min ← MAXINT
    suffix_min ← buffer of size w filled with MAXINT
    j ← 0                                              // Current index in the buffer.
    for i in {0, ..., |vals| − 1} do
        prefix_min ← min(prefix_min, vals[i])
        suffix_min[j] ← vals[i]                        // Write the current value into the buffer.
        j ← j + 1
        if j = w then                                  // When the buffer is full, recompute suffix minima.
            j ← 0
            prefix_min ← MAXINT
            for k in {w − 2, w − 3, ..., 0} do
                suffix_min[k] ← min(suffix_min[k], suffix_min[k + 1])
        if i ≥ w − 1 then                              // Skip the first w − 1 incomplete windows.
            yield min(prefix_min, suffix_min[j])
```

8.2.2. Combining hashes and positions

Since we want to return the leftmost position of the minimum and not the minimum itself, we need to keep track of the positions alongside values. While we could simply store value-position tuples in the buffer, what we do in practice is that we only use the upper 16 bits of each hash value and store the position in the lower 16 bits. This way, taking the minimum still prioritizes the smallest values and breaks ties in favor of the leftmost occurrence as expected. Since the position is stored using 16 bits, strings longer than 2^{16} characters are processed in chunks of 2^{16}. We note here that while using a 16-bit hash is usually not sufficient for k-mer indexing purposes, this *is* sufficiently good for selecting the minimum of a window when $w \ll 2^{16}$, which is typically the case in bioinformatics applications. We intentionally do not expose this hash to the user,

and recommend instead to compute a second larger hash for indexing purposes if needed.

8.3. Computing canonical minimizers

The algorithm we described so far does not take reverse-complement into account. However, in many applications, we want a sequence and its reverse-complement to output the same result. To satisfy this constraint, *canonical minimizers* should return the same set of k-mers regardless of the orientation of the input. Specifically, if the canonical minimizer of a window W is at position p, then the canonical minimizer of the reverse-complement of W should be at position $|W| - k - p = w - 1 - p$. In practice, canonical minimizers are often overlooked as an implementation detail and most existing methods simply compare canonical k-mers, computed as $\hat{x} = \min(x, \mathrm{rc}(x))$, which gives a weaker guarantee [MEK24].

The first step is to adapt the hash function so that a k-mer and its reverse-complement are always hashed to the same value. For rolling hashes, this can easily be achieved using the method described in Section 7.2.3.

Following [PR24], we define the *canonical strand* for each window of length $\ell = w + k - 1$ as the strand where the count of `GT` bases[2] (encoded as `11` and `10` respectively) is the highest, as shown in Algorithm 8.5. This count is in $[0, \ell]$ and when ℓ is odd there can be no tie between the two strands. In code, we instead compute the more symmetric $\#\texttt{GT} - \#\texttt{AC} = 2\#\texttt{GT} - \ell$, which is in $[-\ell, \ell]$, so that count > 0 defines the canonical strand. Then, we select the leftmost forward minimizer when the window is canonical, and the rightmost reverse-complement minimizer when the window is not canonical.

Algorithm 8.5: Counting $\#\texttt{GT} - \#\texttt{AC}$. It can never be 0 when ℓ is odd, and a window is canonical if this is positive.

```
function CANONICAL(w, k, in_out) // in_out iterates pairs (last bp, first bp) of each window.
    ℓ ← w + k − 1                    // Window length, it must be odd to guarantee canonicity.
    c ← −ℓ   // Count #GT − #AC, starting at −ℓ and adding/subtracting 2 for each G or T.
    for (in_bp, _) in in_out[0 : ℓ − 1] do    // The first ℓ − 1 iterations, there is no out_bp.
        c ← c + (in_bp & 2)
    for (in_bp, out_bp) in in_out[ℓ − 1 :] do   // 2-bit bp coming in and out of each window.
        c ← c + (in_bp & 2)
        yield c > 0                          // A window is canonical if #GT > #AC.
        c ← c − (out_bp & 2)
```

The benefit of this method over, say, determining the strand via the middle character (assuming again that ℓ is odd), is that the `GT` count is more stable across consecutive windows, since it varies by ± 1. This way, the strandedness and thus the chosen minimizer is less likely to flip between adjacent windows.

One small drawback of this scheme is that it is not *forward*, i.e. the selected positions are not guaranteed to be non-decreasing. Suppose for instance that a long window has many occurrences of the smallest minimizer, and that shifting the window one position changes its canonical strand. In that case, the position of the sampled minimizer could jump backwards: from sampling the rightmost minimizer to sampling the leftmost minimizer. In practice, this does not seem to be a major limitation, both because it is rare and because downstream methods usually work fine on non-forward schemes anyway. Another (slower) scheme for canonical minimizers preserving the forward property is described in Appendix A.

[2] Any pair of non-complementary bases would work here.

8.4. Vectorized implementation

8.4.1. Parallel iteration and sliding window minimum

Since computing a sliding window minimum is inherently sequential, we reuse the approach presented in Section 7.3 for parallel iteration. This way, we iterate over the hashes of L (the number of lanes) independent chunks, and we can compute their sliding window minimum simultaneously. The only difference here is that the chunks must overlap by $w + k - 2$ bases (instead of $k - 1$ for hashes alone) since we want to cover all windows of w k-mers.

The scalar version of the two-stacks method presented in Algorithm 8.4 can easily be adapted to use vectorized instructions. Algorithm 8.6 shows a vectorized version specialized for 8 lanes of 32-bit hashes, which keeps track of the position in the lower 16 bits.

Algorithm 8.6: SIMD two-stacks algorithm for sliding window minimum, with leftmost tie-breaks.

```
function SimdSlidingMin(w, hash_it) // hash_it iterates over 32-bit hashes of each k-mer.
    val_mask ← 0xffff0000                                  // Use high 16 bits of each value.
    pos_mask ← 0x0000ffff                                  // Low 16 bits store positions.
    prefix_min ← 0xffffffff
    suffix_min ← buffer of size w filled with 0xffffffff
    n ← |hash_it| − w − 1                  // The first w − 1 hashes do not complete a window.
    i ← LANE_IDX × n                                       // Current position of the lane.
    j ← 0                                                  // Current index in the buffer.
    for h in hash_it do
        val ← (h & val_mask) | i
        prefix_min ← min(prefix_min, val)
        suffix_min[j] ← val                                // Write the current value into the buffer.
        j ← j + 1
        if j = w then                                      // When the buffer is full, recompute suffix minima.
            j ← 0
            prefix_min ← 0xffffffff
            for k in {w − 2, w − 3, ..., 0} do
                suffix_min[k] ← min(suffix_min[k], suffix_min[k + 1])
        if i ≥ LANE_IDX × n + w − 1 then                   // Skip the first w − 1 incomplete windows.
            yield min(prefix_min, suffix_min[j]) & pos_mask
        i ← i + 1
```

8.4.2. Deduplication

The last step of the minimizer algorithm is to *deduplicate* the results, since adjacent windows often share the same minimizer position. In practice, deduplicating works best when the data to be deduplicated is linear in memory. But the output of the vectorized sliding-window minimum gives a `u32x8` containing the position of one minimizer of each chunk. Thus, every L iterations we reuse the matrix transpose (discussed in Section 7.3.1) to obtain a `u32x8` for each chunk, containing L consecutive minimizer positions.

We deduplicate each lane using the technique of Lemire [Lem17]. This compares each element to the previous one, and compares the first element to the last minimizer of the window before. The distinct elements are then *shuffled* to the front of the SIMD vector using a lookup table and appended to a buffer for each lane, as illustrated in Figure 8.3. We end by concatenating all the per-lane buffers into a single vector of minimizer positions, and make sure to avoid duplicates between the end and start of adjacent lanes.

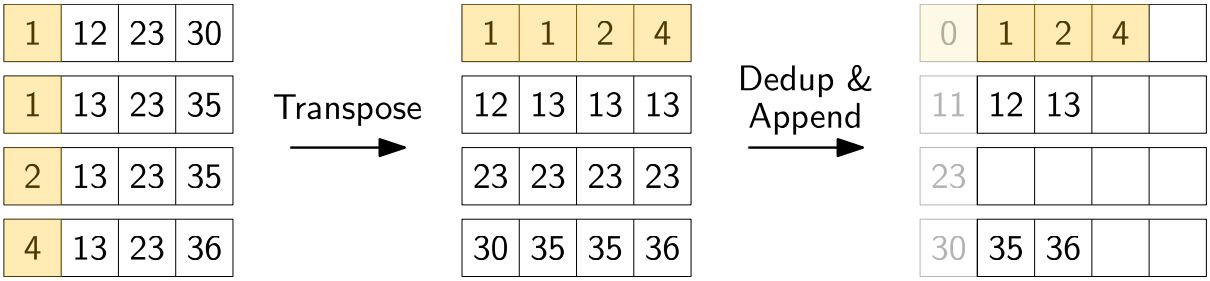


Figure 8.3.: Overview of the reordering and deduplication of minimizer positions. Each row on the left corresponds to the minimizer positions outputted by the SIMD sliding window minimum algorithm for L chunks at a time. We first transpose this matrix, so that the resulting SIMD vectors each correspond to a single chunk, with values corresponding to the first chunk highlighted in yellow. These are then compared with their preceding element (including the previous minimizer position, as shown in gray), and distinct elements are shuffled to the front. These positions are accumulated in a separate buffer for each chunk, and these buffers are finally concatenated into a single flat vector. Note that while the figure shows $L = 4$ lanes, we use $L = 8$ lanes in practice for 256-bit registers.

8.5. Variations of minimizers

While the method we presented so far is designed to compute minimizer positions, we can easily adapt it for different variations of the problem.

8.5.1. Computing super-*k*-mers

One of the most common use cases of minimizers is to partition a sequence into super-k-mers (presented in Chapter 4). In that case, in addition to the minimizer positions, we also want to keep track of which windows are associated to which minimizers. A simple way to encode this information is to record the starting position of the first window with a given minimizer position.

The starting position of super-k-mers can be extracted by amending the deduplication step. After comparing adjacent minimizer positions, we obtain a mask that determines the shuffle instruction to apply. Normally we shuffle the 32-bit minimizer positions directly. Instead, we can mask out the upper 16 bits (previously used to store hashes) and store there the index of its window. If we then shuffle those values, we obtain for each minimizer its position in the input text, and the position of the first window where this k-mer became a minimizer. This information is sufficient to recover all super-k-mers, and sequences longer than 2^{16} bp can simply be split up.

8.5.2. Computing syncmers

Another frequent use case is to compute open/closed syncmers (also presented in Chapter 4). For closed syncmers, we want to output windows starting/ending with their minimizer, while for open syncmers, we want to output those with a minimizer in the middle position (assuming w is odd).

Computing syncmers can be implemented as a simple *filter* on top of the *non-deduplicated* minimizer positions. For each window at position i, we test whether the minimizer position p satisfies the syncmer condition: $p = i$ or $p = i + w - 1$ for closed syncmers, or $p = i + \lfloor w/2 \rfloor$ for open syncmers (assuming w is odd so that the middle element is unique). This test can be performed efficiently using vectorized equality comparisons: non-syncmer positions are erased via a *blend* instruction (which branchlessly selects values based on a boolean mask), and the

same shuffle-append mechanism used in Section 8.4.2 efficiently collects the remaining positions. Note that the deduplication step is not needed here since syncmer positions are always distinct.

8.5.3. Other minimizer schemes

Finally, we can change the implementation of the minimizer scheme itself. First, if we want to change the order on k-mers, we can replace the rolling hash by the function of our choice. For instance, we can simply use the identity for a lexicographic order, or flip the first bits for an anti-lexicographic order [Gro25a].

Additionally, we can tweak the sliding window scan to implement other low-density schemes (further discussed in Chapter 15). The best example is probably mod-minimizers [GP24], which only require applying a modulo w on the position of the minimum as an extra step. Note however that applying a modulo on vector registers is quite impractical, meaning that we prefer subtracting and blending results in practice.

8.6. Experimental evaluation

Our vectorized implementation, built on top of packed-seq and seq-hash, is available at https://github.com/rust-seq/simd-minimizers. This implementation supports canonical minimizers, super-k-mers and syncmers computation, and works with both AVX2 and NEON. The code to reproduce the experiments is available in the `bench` folder of the same repository.

The experiments were run on an Intel Core i7-10750H with AVX2 running at 2.6GHz and an Apple M1 with NEON running at 3.2GHz, using Rust nightly 1.88.0. As input, we use a fixed random string of 10^8 bases, encoded either as ASCII or as packed representation depending on the method. Reported timings are the median of five runs and shown in nanoseconds per base.

In our experiments, we use parameter values for w and k as used by Kraken2 [WLL19] (5, 31), SSHash [Pib22] (11, 21), and minimap2 [Li18] (19, 19).

8.6.1. Incremental time usage

Table 8.2 shows the time usage for various incremental subsets of our method.

Table 8.2.: Total time per base taken after each incremental step, for $(w, k) = (11, 21)$.

Step	ns/bp AVX2	NEON
Iterate the bases	0.15	0.10
+ collect to vector	0.30	0.18
+ iterate the delayed bases	0.30	0.19
+ ntHash	0.32	0.33
+ sliding window min	0.90	0.82
+ collect	1.48	0.95
+ dedup	1.61	1.38
+ canonical ntHash	1.04	0.85
+ canonical strand	1.53	1.16
+ collect	2.03	1.37
+ dedup	2.20	1.82

To start with, iterating the 8 chunks of the input and summing all bases takes 0.15 ns/bp (0.10 ns/bp on NEON). Appending all `u32x8` SIMD vectors containing the bases to a vector takes 0.30 ns/bp (0.18 ns/bp on NEON), indicating that writing to memory induces some overhead.

Collecting the second $k-1$-delayed stream of characters that leave the k-mer (by adding them to the non-delayed stream) has no additional overhead. Computing ntHash only takes 0.02ns/bp extra on AVX2, but adds 0.15 ns/bp on NEON. The sliding window computation nearly triples the total time. Collecting the minimizer positions to a linear vector (i.e., transposing matrices and writing output for each of the 8 chunks) again incurs 50% overhead, and deduplicating them again adds some time.

Going back a step, using canonical ntHash instead of forward ntHash takes 0.14 ns/bp extra on AVX2 but only 0.03 ns/bp on NEON, and determining the canonical strand (via a third ℓ-delayed stream and counting `GT` bases) takes another 0.49 ns/bp on AVX2 and 0.31 ns/bp on NEON. As before, collecting and deduplicating are slow and add around 0.70ns/bp.

In conclusion, we see that iterating the chunks of the input and determining minimizers is quite fast, but that a lot of time must then be spent to "deinterleave" the output into a linear stream. As can be expected, canonical minimizers are slower to compute than forward minimizers, but the overhead is less than 50%, which seems quite low given that the ntHash and sliding window minimum computation are duplicated and a canonical-strand computation is added.

8.6.2. Full comparison

We compare against the `minimizer-iter` crate v1.2.1[3], which implements a queue-based sliding window minimum using `wyhash`[4] and also supports canonical minimizers. For an additional comparison, we optimized an implementation of the remap method with ntHash based on a code snippet by Daniel Liu[5]. Results are in Table 8.3 and Figure 8.4.

`minimizer-iter` takes around 26 ns/bp for forward and 33 ns/bp for canonical minimizers, and its runtime does not depend much on w and k, because the popping from the queue is unpredictable regardless of w. Rescan starts out at 11.7 ns/bp for $w = 5$ and gets significantly faster as w increases, converging to around 5 ns/bp for $w \gg 100$. This is explained by the fact that rescan has a branch miss every time the current minimizer falls out of the window, which happens for roughly half the minimizers at a rate of $1/(w+1)$. Thus, as w increases, the method becomes more predictable and branch misses go down. Our method, `simd-minimizers` runs around 1.61 ns/bp for forward and 2.20 ns/bp for canonical minimizers when given packed input, and therefore is 3.4× to 6.8× faster than the rescan method.

In Figure 8.4, we see that `simd-minimizers` performance is mostly independent of k and w since it is mostly data-independent. Only for small $w \leq 5$ it is slightly slower due to the larger number of minimizers and hence larger size of the output.

As we use SIMD with 8 lanes, we could in theory expect up to 8× speedup. In practice this is hard to reach because of constant overhead and because the overhead to work well with SIMD in the first place. In particular for large w, rescan benefits from very predictable and simple code and only outputs unique minimizer positions, making it very efficient. In SIMD, on the other hand, we use a data-independent algorithm, and output the minimizer position for every single window, which then has to be deduplicated. Thus, it is nice to see that even for large w, our method is over 3× faster, despite this overhead.

8.6.2.1. ASCII input and mulHash

Apart from taking bit-packed input, `simd-minimizers` also works on ASCII-encoded DNA sequences of `A/C/T/G` characters directly, which are then packed into values $\{0, 1, 2, 3\}$ for ntHash (in that order) during iteration. This is around 0.25ns/bp slower, mostly because of the larger size of the unpacked input.

[3] https://github.com/rust-seq/minimizer-iter
[4] https://github.com/eldruin/wyhash-rs
[5] https://gist.github.com/Daniel-Liu-c0deb0t/7078ebca04569068f15507aa856be6e8

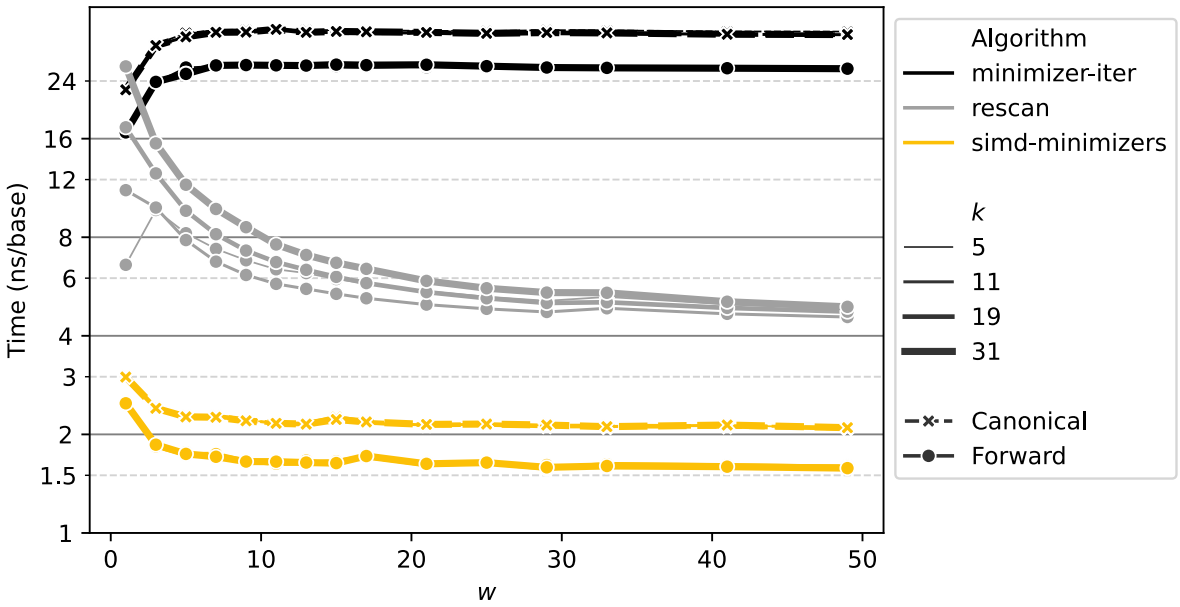
Table 8.3.: Comparison of `simd-minimizers` against `minimizer-iter` and a rescan implementation. Times in ns/bp for forward and canonical minimizers, with various (w, k) tuples.

(a) Comparison on AVX2.

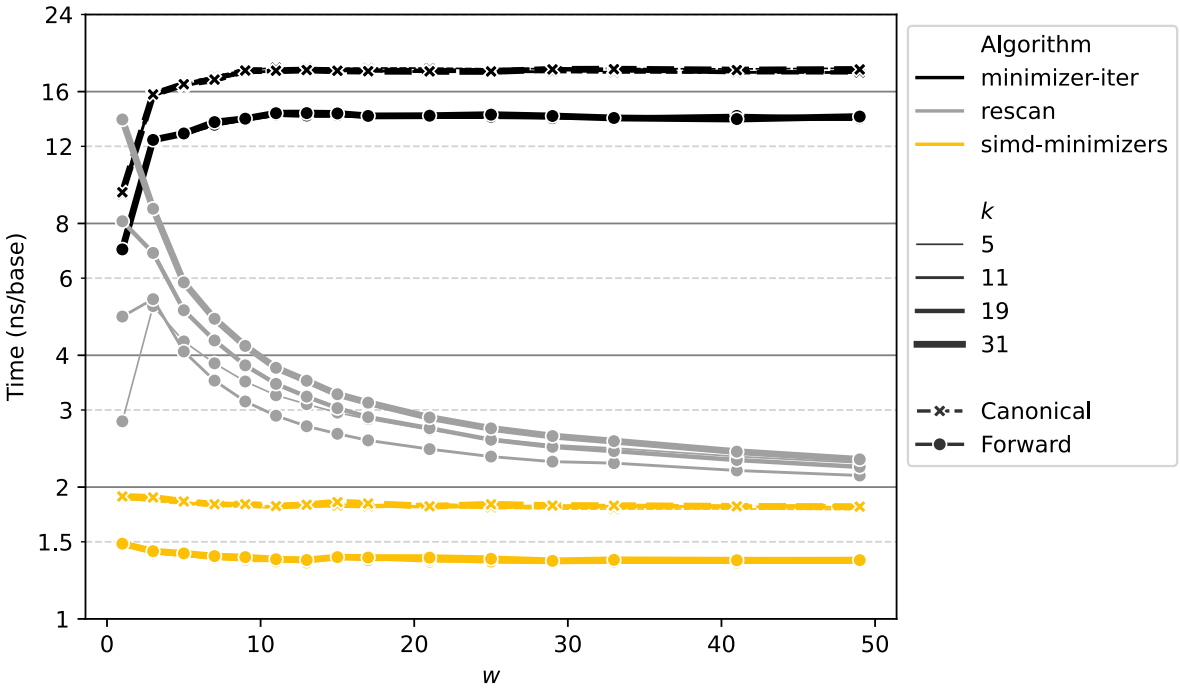
Method	(w=5, k=31)		(w=11, k=21)		(w=19, k=19)	
	fwd.	canon.	fwd.	canon.	fwd.	canon.
`minimizer-iter`	25.30	32.84	26.96	33.93	26.81	34.04
Rescan ntHash	11.65		7.41		5.61	
`simd-minimizers` ntHash						
- packed input	1.69	2.28	1.61	2.20	1.64	2.16
- on-the-fly packing	1.92	2.50	1.84	2.42	1.91	2.42
Rescan mulHash	11.37		6.79		5.76	
`simd-minimizers` mulHash						
- packed input	1.85	2.49	1.74	2.40	1.78	2.42
- on-the-fly packing	2.12	2.70	2.05	2.62	2.05	2.65
- ASCII input	2.11	2.71	2.06	2.63	2.01	2.66

(b) Comparison on NEON.

Method	(w=5, k=31)		(w=11, k=21)		(w=19, k=19)	
	fwd.	canon.	fwd.	canon.	fwd.	canon.
`minimizer-iter`	12.89	16.60	14.26	18.08	14.20	18.04
Rescan ntHash	5.95		3.92		2.82	
`simd-minimizers` ntHash						
- packed input	1.42	1.85	1.38	1.82	1.38	1.83
- on-the-fly packing	1.66	2.07	1.58	1.99	1.59	2.02
Rescan mulHash	6.94		3.93		3.42	
`simd-minimizers` mulHash						
- packed input	1.97	4.34	1.91	4.27	1.93	4.26
- on-the-fly packing	2.11	4.37	2.04	4.41	2.03	4.30
- ASCII input	2.12	4.39	2.06	4.44	2.04	4.29



(a) Running time on AVX2.



(b) Running time on NEON.

Figure 8.4.: Running time (logarithmic) of `minimizer-iter`, rescan, and `simd-minimizers` for different values of w and k.

The mulHash variant is around 0.20ns/bp slower again, but works for any ASCII input. Performance on 100 MB of the Pizza&Chili corpus English[6] and Sources[7] datasets is nearly identical to performance on the random DNA shown in Table 8.3.

8.6.2.2. Human genome

We also run `simd-minimizers` on the chromosomes of the T2T-CHM13v2.0 human genome[8] [Nur+22], of total size 3.2 Gbp. Here, computing forward minimizers takes 5.19 seconds, and canonical minimizers takes 6.71 seconds for $(w, k) = (11, 21)$, which corresponds to 1.67 and 2.15 ns/bp, which is within a few percent of Table 8.3.

8.6.2.3. Density

For $(w, k) = (11, 21)$, we get a density of 0.173 for forward minimizers, and 0.167 for canonical minimizers, which are both close to the expected density of $2/(w+1) = 1/6 \approx 0.167$. Changing to $(w, k) = (19, 19)$, we get density 0.098 for forward minimizers and 0.100 for canonical minimizers, both close to the expected density of $2/(19 + 1) = 1/10 = 0.1$. Thus, we see that in practice, ntHash is a sufficiently random hash function to approach the expected density of random minimizers with a perfectly uniform random hash function.

8.6.2.4. Multithreading

We test throughput in a multithreaded setting, by using 6 threads to process the chromosomes in parallel. This way, processing takes 0.97 s (forward) and 1.27 s (canonical) when the input data is already loaded into memory, showing slightly above 5× speedup. This is just below 6× speedup as the chromosomes don't perfectly partition the data into 6 equal parts, so that some threads finish before others. For applications, we recommend to simply call our library in parallel from multiple threads as needed.

[6] https://pizzachili.dcc.uchile.cl/texts/nlang/
[7] https://pizzachili.dcc.uchile.cl/texts/code/
[8] Available at https://github.com/marbl/CHM13.

9. Application to sequence filtering

> **Note**
>
> This chapter is adapted from Martayan et al. (2026), published in Peer Community Journal.

The previous chapters of this part developed efficient algorithms for parsing sequences (Chapter 6), hashing k-mers (Chapter 7), and computing minimizers at high throughput (Chapter 8). This chapter presents an application of these building blocks to *sequence filtering*, a problem that arises naturally at the boundary between large-scale k-mer indexing and exact sequence analysis.

Large-scale k-mer indexes can efficiently identify a small, relevant subset of documents (e.g., read sets, unitigs, contigs). However, this initial filtering is not final. Due to false positives from probabilistic indexes, or sparse/non-collinear matches, queried k-mers may not be truly present or usable for mapping. Therefore, recovering specific sequences within identified documents that genuinely share k-mers with the query is crucial. This refinement, moving from relevant documents to relevant sequences suitable for alignment, mitigates downstream scalability issues and can reduce data volume by orders of magnitude, such as identifying a few relevant sequences from billions.

The problem of k-mer-based *sequence filtering* can be formalized as follows: given a set of k-mers of interest $\mathcal{Q} = \{q_1, \dots, q_m\}$ (the "queries"), a set of sequences $\mathcal{S} = \{S_1, \dots, S_n\}$ and a threshold T, output for each $S_i \in \mathcal{S}$ whether $\sum_{q \in \mathcal{Q}} \#\text{occ}(q, S_i) \geq T$, i.e. whether S_i contains at least T occurrences of k-mers of interest. In practice, the threshold can either be set as an absolute value $T \in \mathbb{N}$ or as a relative value proportional to the size of the sequence: given $t \in [0, 1]$, $T_i = \lceil t \cdot (|S_i| - k + 1) \rceil$.

Related work

Although sequence-level indexes have been proposed [KMRK22; VCL25], indexing and storing entire dataset collection at this resolution remains expensive and challenging at scale, thus favoring lightweight filtering methods directly from unindexed data. Efficient multi-pattern matching is a highly studied topic from both the theoretical [Cro+99] and practical points of view [FL13]. Practical implementation can be found in classical data processing with grep-like efficient tools such as Ripgrep [Gal24] or Hyperscan [Wan+19]. Some tools perform pattern matching adapted to genomic data, such as Seqkit [SLLH16; SSZ24], fqgrep [Hom+25] or grepq [Cro25]. However, these solutions tend to scale poorly as the number of query k-mers increases and fail to leverage the fixed length of the search patterns. This is a critical issue, as querying numerous k-mers is often necessary for large queries (contigs, long reads, whole genomes), searching for many distinct sequences (variant or gene collections), or optimizing batched queries. The recent BackToSequences [BMAP24] tool, used in the Logan project, addresses these issues by simply indexing query k-mers in a hash table to search for target documents. Very recently, both Deacon [CLC25] and Cleanifier [ZSR25] were proposed as scalable solutions for contaminant depletion, that is removing sequences originating from a given genome, which is a direct application of sequence filtering. In particular, Deacon effectively uses many of the techniques presented in this chapter. Finally, while to our knowledge it has not been used for sequence filtering, the spectral Burrows-Wheeler transform (SBWT) [APV23] is also a suitable method for this problem thanks

to efficient matching statistics [MABP25].

Contributions

This work presents three main contributions. First, we introduce a new algorithm that uses random minimizers to quickly filter out sequences that contain too few k-mers of interest, effectively reducing the cost of negative matches by a factor $w/2$. Second, we propose an optimized implementation named *K2Rmini*, that takes advantage of vectorized instructions to parse sequences, hash k-mers and compute minimizers efficiently. Third, we compare our approach to many different tools and evaluate their scalability for a large number of patterns, highlighting the benefits of different methods. Our tool, K2Rmini, is able to filter 2 Gbp/s on a consumer laptop and is available online at https://github.com/Malfoy/K2Rmini.

9.1. Filtering sequences with minimizers

Classical multi-pattern matching algorithms, which generalize the Knuth-Morris-Pratt approach, construct deterministic finite automata to achieve linear time complexity relative to the pattern and input data sizes. While these methods offer strong theoretical guarantees, they often exhibit poor cache locality. Furthermore, they address the more general problem of matching variable-length patterns, whereas our work focuses exclusively on fixed-size patterns. This constraint creates an opportunity for hash-based techniques, which are better suited for large pattern sets and more amenable to hardware acceleration through vectorized instructions.

One of the state-of-the-art solutions, BackToSequences [BMAP24], uses an efficient hash table containing all k-mers of interest and skips irrelevant fields. However, its main limitation is its necessity to perform a hash table lookup for every k-mer in the data stream. This process becomes computationally expensive, especially when the hash table is large and does not fit into low-level caches. To address this bottleneck, we propose K2Rmini (https://github.com/Malfoy/K2Rmini), a method that integrates minimizer-based filtering and Single Instruction, Multiple Data (SIMD) operations.

9.1.1. Using minimizers to upper bound the number of k-mer matches

> Different notations
>
> We reuse the notations from Section 4.3: the minimizer size is denoted m while k denotes the number of bases in a window, i.e. $k = w + m - 1$.

As discussed in Chapter 4 and Chapter 8, random minimizers select the smallest m-mer in each window of w consecutive k-mers according to a random order and guarantee local consistency: two sequences sharing a context of w k-mers always produce the same minimizer, making them a useful proxy for k-mer overlap [Li18; Dar+22; Eki+23].

One core idea of our approach is to leverage this property to avoid exhaustive k-mer scanning. Given a set $\mathcal{Q}$ of k-mers of interest, we compute the set of associated minimizers $\mathcal{M}(\mathcal{Q}) = \{\text{minimizer}(q) : q \in \mathcal{Q}\}$. Then, instead of checking every k-mer from a sequence S, we only compare the minimizers of S against $\mathcal{M}(\mathcal{Q})$. We know that each minimizer match implies that up to w k-mers of S are in $\mathcal{Q}$. Thus, if we have ℓ minimizer matches, the number of k-mer matches is upper-bounded by $\ell \times w$, so having $\ell < \lceil \frac{T}{w} \rceil$ is sufficient to discard a sequence.

While this upper bound is quite accurate for sparse matches, it is very loose (by a factor close to 2) for dense matches. One way to refine this upper bound is to observe that if two minimizers are d positions apart from each other, they cover at most $w + d$ k-mers. Therefore, if we have a

chain of consecutive minimizer matches at positions $p_1, \dots, p_r$, the number of k-mer matches is upper-bounded by $w + p_r - p_1$.

In practice, we often have access the exact number of k-mers covered by each minimizer as a byproduct of the sliding window algorithm [GM25]. In that case, we can simply sum them to get a tight upper bound.

9.1.2. Minimizer-based filtering algorithm

Our algorithm, detailed in Algorithm 9.1, works in two passes. The first pass computes an upper bound on the number of k-mer matches, as detailed in the previous section, by performing $\mathcal{O}\left(\frac{|S|}{w}\right)$ queries to $\mathcal{T}_{\mathcal{M}}$ (the hash table of minimizers). This has two main advantages: not only do we reduce the number of lookups, but querying $\mathcal{T}_{\mathcal{M}}$ is also faster than $\mathcal{T}_{\mathcal{Q}}$ (the hash table of k-mers) because it is roughly $w/2$ times smaller when k-mers from $\mathcal{Q}$ are consecutive.

The second pass only happens for sequences with an upper bound on k-mer matches above the threshold. In that case, we must count the exact number of k-mer matches to identify true hits. Deacon [CLC25], another tool tailored for decontamination developed by Bede Constantinides, uses a simplified version of the method proposed here. In particular, it omits this second pass, reducing computational overhead at the cost of potential false positives.

Algorithm 9.1: Minimizer-based filtering.

```
function FILTER(set of k-mers Q, set of sequences S, threshold T)
    T_Q ← HashSet(Q)                          // hash table of k-mers of interest
    T_M ← HashSet(M(Q))                       // hash table of minimizers of interest
    for S ∈ S do
        u ← 0                                 // upper bound of k-mer matches
        for each minimizer x in S, covering c k-mers do
            if x ∈ T_M then
                u ← u + c
        if u < T then
            yield false
            continue
        s ← 0                                 // exact count of k-mer matches
        for each k-mer x in S do
            if x ∈ T_Q then
                s ← s + 1
        yield s ≥ T
```

9.1.3. Vectorization

Multiple parts of our algorithm are accelerated using vectorized instructions. For sequence parsing, we use the `helicase` library presented in Chapter 6, which outputs bitpacked representations for faster downstream processing. Minimizer positions and k-mer coverage counts are computed with SimdMinimizers (discussed in Chapter 8). Finally, k-mer lookups in the second pass are accelerated with the vectorized rolling hash presented in Chapter 7.

9.1.4. Parallelization

On top of the vectorized computation of the matches, we adopt a parallelization strategy for the main loop based on a producer-consumer model. A producer thread parses the sequences and sends batches of sequences (up to a desired length) to multiple consumer threads, which in turn compute the matches on their batch and output the results. One approach that we have not

implemented yet but could be worth trying would be to let each consumer thread directly parse a chunk of the input file in parallel, thus removing the need for a producer thread and reducing the communication overhead.

9.2. Results

All experiments were run with the benchmarking workflow available at https://github.com/imartayan/K2Rmini_experiments. The benchmarking framework is easily extensible, since each additional program only requires a dedicated command wrapper to be added to a common interface before it can be executed and plotted with the other tools. All reported times include both the cost of indexing the queried patterns and the filtering step. Benchmarks were performed on a dual-socket Intel Xeon Gold 6430 machine with 64 cores and 512 GB of RAM running Ubuntu 24.04.2. Because no ready-to-use SBWT-based sequence filtering implementation was available for our setting, we implemented the SBWT baseline ourselves[1] using the Rust `sbwt` crate[2].

9.2.1. State-of-the-art analysis

To evaluate state-of-the-art tools, we benchmarked a selection of specialized and general-purpose programs while varying the number of queried k-mers. The specialized tools included Seqkit 2.13.0 [SLLH16; SSZ24], fqgrep 1.1.1 [Hom+25], grepq 1.6.5 [Cro25], BackToSequences 0.8.3 [BMAP24], Cleanifier 1.2.0 [ZSR25], Deacon 0.14.0 [CLC25], and our SBWT-based implementation. For comparison, we also included grep 3.11, ripgrep 15.1.0 [Gal24], and Hyperscan 5.4.2 [Wan+19] (or Vectorscan 5.4.12 on ARM). As shown in Figure 9.1, Figure 9.2, we benchmarked these tools on 2.6 Gbp of PacBio HiFi reads from HG002[3] while increasing the number of queried k-mers.

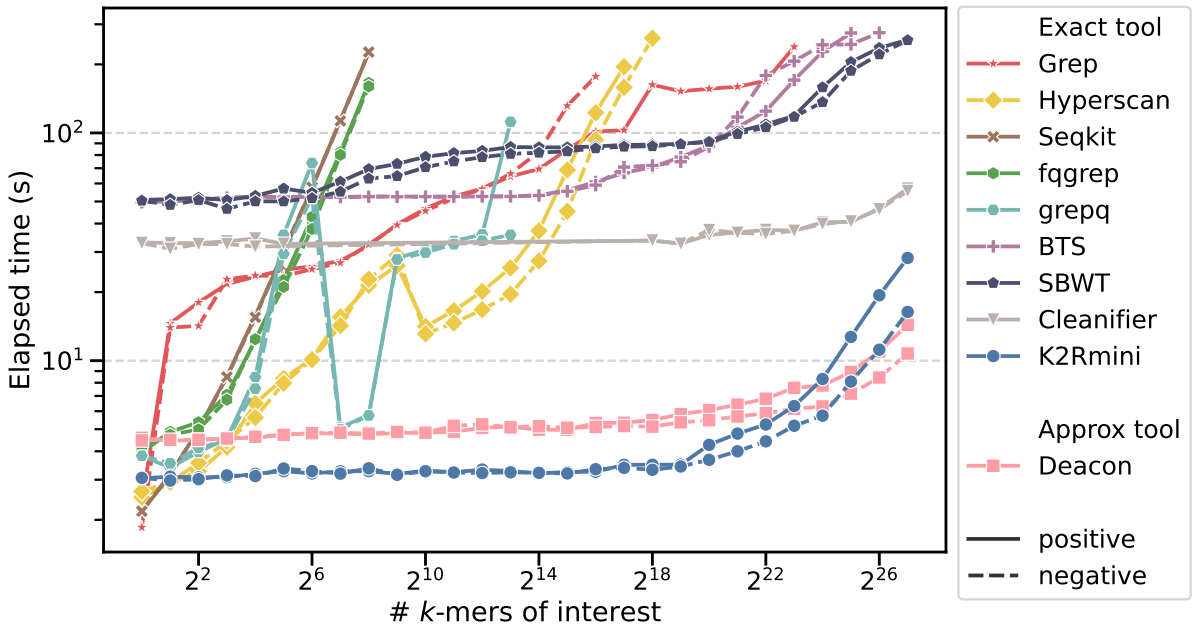
The results separate the tested methods into two broad groups. Classical multiple-pattern matching tools such as grep, ripgrep, Hyperscan, Seqkit, fqgrep, and grepq scale poorly as the number of queried k-mers increases. Their running time rises rapidly and they become impractical once the query set reaches the large regimes that matter for genomic filtering. In contrast, methods that explicitly index the queried patterns, or a sketch derived from them, remain effective over the full range of tested query sizes.

Among these scalable approaches, Deacon and K2Rmini are the fastest overall. Cleanifier is also largely insensitive to the number of queried k-mers, but with a clearly higher absolute running time. SBWT and BackToSequences remain much more scalable than general-purpose pattern matchers, although both become slower than Deacon and K2Rmini as the number of queried k-mers grows. In the multithreaded setting, SBWT remains consistently ahead of BackToSequences for the largest query sets, while both are clearly outperformed by the minimizer-based approaches.

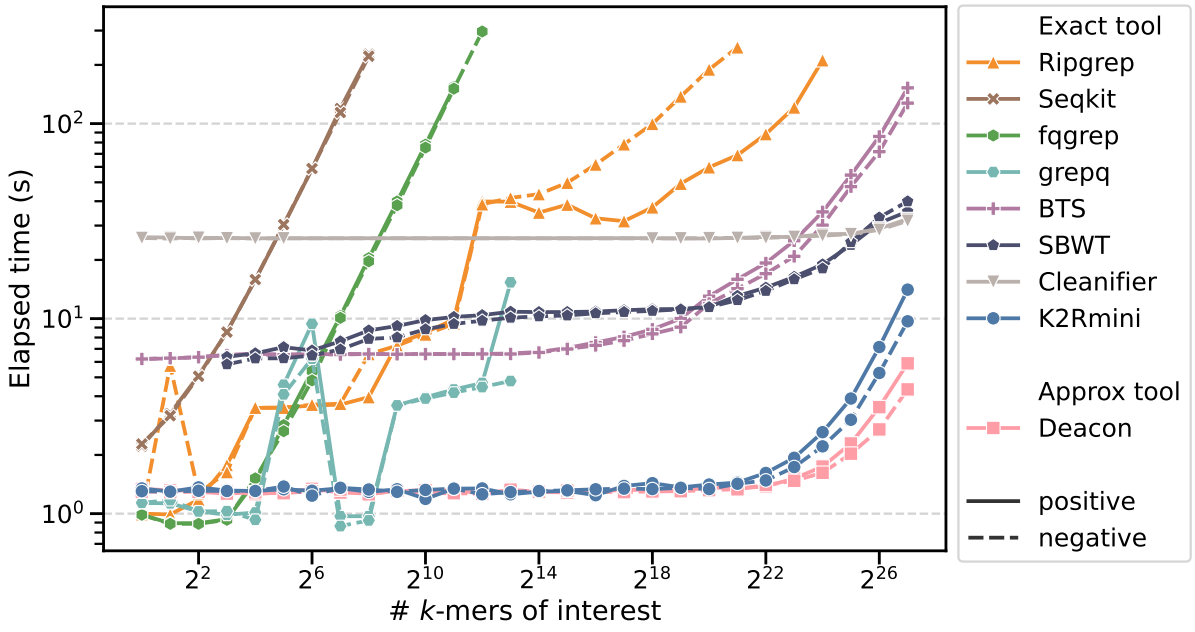
The gap between positive and negative queries is especially informative for K2Rmini. For negative k-mers, most reads are rejected during the minimizer pass, which keeps the running time close to flat across a broad range of query counts. For positive k-mers, more reads pass this first filter and trigger the exact counting phase, so the running time increases more substantially. This behavior directly reflects the design of K2Rmini, where minimizers are used to avoid unnecessary exact searches, but exact verification is still required whenever the upper bound is above threshold.

This comparison must be interpreted more carefully for Deacon. Deacon only indexes minimizers and does not perform an exact second pass on the surviving reads. Its excellent scalability

[1] https://github.com/imartayan/K2Rmini_experiments/tree/main/sbwt_filter
[2] https://docs.rs/sbwt/latest/sbwt/
[3] https://s3-us-west-2.amazonaws.com/human-pangenomics/NHGRI_UCSC_panel/HG002/hpp_HG002_NA24385_son_v1/PacBio_HiFi/15kb/m54328_180928_230446.Q20.fastq



(a) Running time with a single thread.



(b) Running time with 8 threads.

Figure 9.1.: Running time of state-of-the-art tools when filtering 2.6 Gbp of PacBio HiFi reads from HG002 (`m54328_180928_230446`) with an increasing number of queried k-mers for $k = 31$. Positive k-mers are extracted from the reads while negative k-mers are random sequences.

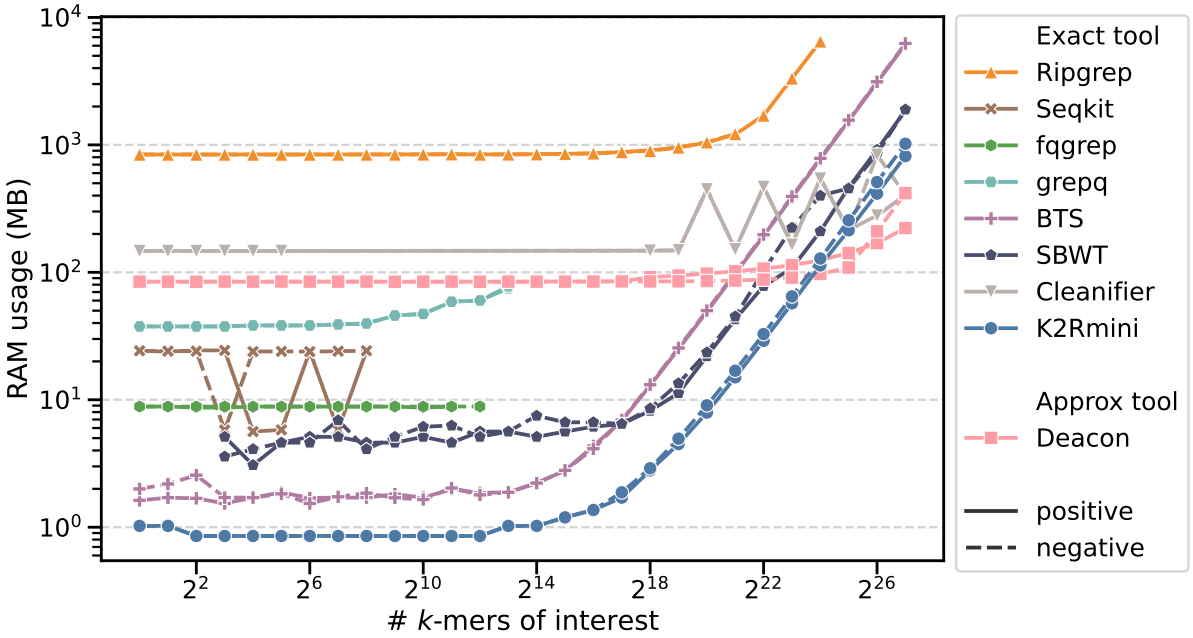


Figure 9.2.: Memory usage of state-of-the-art tools when filtering 2.6 Gbp of PacBio HiFi reads from HG002 (`m54328_180928_230446`) with an increasing number of queried k-mers for $k = 31$. Positive k-mers are extracted from the reads while negative k-mers are random sequences.

therefore comes from solving a relaxed version of the problem, which may introduce false positives. By contrast, K2Rmini uses minimizers as a lossless prefilter and performs exact k-mer counting whenever necessary.

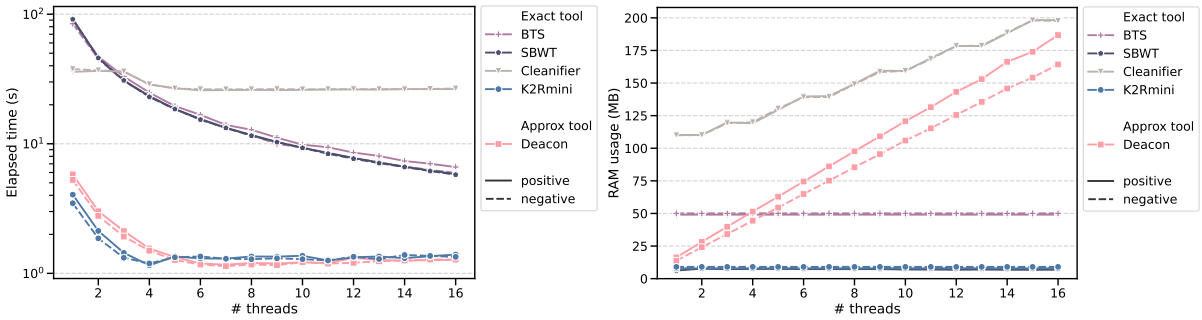
A practical observation is that at some point, pattern indexing itself is a visible part of the total running time and filtering is no longer the only bottleneck.

Memory usage reveals a similar separation between approaches. General-purpose tools such as ripgrep already have a large baseline memory footprint and can reach several gigabytes for the largest query sets. By contrast, the specialized genomic tools initially use much less memory, but their scaling behavior differs substantially. Among the scalable exact methods, K2Rmini has the lowest memory footprint over most of the explored range and remains below BackToSequences and SBWT at large query sizes. BackToSequences starts from a very small footprint on small query sets, but its memory usage grows steeply and eventually becomes the highest among the indexed genomic methods. SBWT scales more smoothly than BackToSequences, but still requires substantially more memory than K2Rmini for large pattern sets. Cleanifier and Deacon exhibit a higher baseline memory usage, yet their growth is more moderate, which is consistent with their good scalability in running time. Overall, these results show that K2Rmini provides the best end-to-end compromise among exact methods, combining low running time with low memory usage over a broad range of query sizes.

9.2.2. Influence of parameters

To better understand the behavior of the most scalable approaches, we compared BackToSequences, SBWT, Deacon, and K2Rmini while varying the number of threads and the k-mer size. The results are shown in Figure 9.3, Figure 9.4.

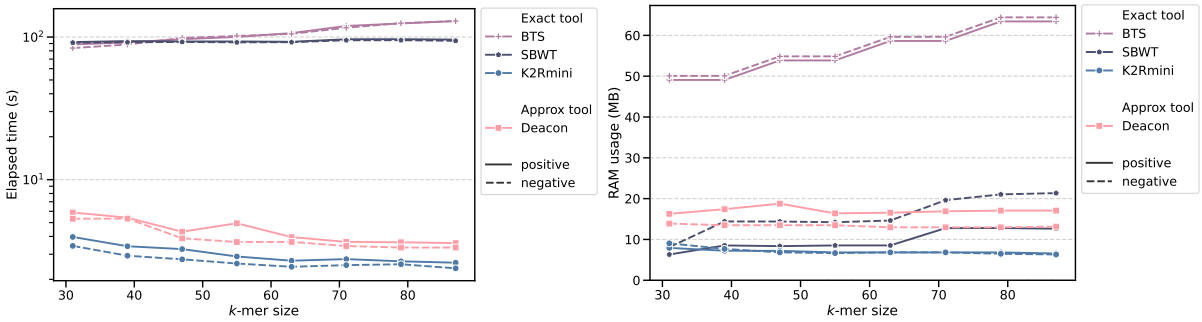
When increasing the number of threads, K2Rmini remains the fastest method throughout the benchmark. Most of the speedup is obtained within the first four threads, after which the running time stabilizes around 1.3–1.4 seconds. Deacon shows a very similar trend, with slightly



(a) Running time versus number of threads.

(b) Memory usage versus number of threads.

Figure 9.3.: Influence of the number of threads when filtering 2.6 Gbp of reads with 1M patterns of size $k = 31$.



(a) Running time versus k.

(b) Memory usage versus k.

Figure 9.4.: Influence of the k-mer size when filtering 2.6 Gbp of reads with 1M patterns and a single thread. Cleanifier is not included because it does not support $k > 32$.

higher running times and the same early saturation. This suggests that, for both minimizer-based approaches, the main bottleneck quickly shifts away from the matching procedure itself toward parsing and other shared overheads. In contrast, BackToSequences and SBWT continue to benefit from additional threads over the full range, with their running times decreasing steadily from about 100 seconds on one thread to around 6 seconds on 16 threads. Cleanifier shows almost no parallel speedup in this experiment, remaining close to 35 seconds regardless of the thread count.

The memory trends differ markedly from the time trends. K2Rmini has the lowest memory footprint and remains nearly constant as the number of threads increases, staying around 8–10 MB. SBWT also remains essentially flat, at about 8 MB. BackToSequences has a larger but still stable footprint, around 50 MB, independent of the number of threads. By contrast, Deacon and Cleanifier both show a substantial increase in memory usage with additional threads. Deacon grows roughly linearly from about 15 MB on one thread to around 180 MB on 16 threads, while Cleanifier increases from about 110 MB to about 195 MB. Therefore, among the tested methods, K2Rmini combines the best running time with the smallest and most stable memory usage in the multithreaded setting.

We also evaluated the effect of the k-mer size with a single thread and 1M queried patterns. As k increases, K2Rmini becomes slightly faster, decreasing from about 4 seconds at $k = 31$ to about 2.3 seconds for the largest tested values. This trend is consistent with our minimizer-based design. Keeping the minimizer size fixed while increasing k enlarges the minimizer window, reducing minimizer density, and therefore decreasing the number of lookups required during the first filtering pass. Deacon exhibits a similar, though less pronounced, decrease in running time. In contrast, BackToSequences becomes slower as k increases, rising from about 80 seconds to more than 120 seconds, while SBWT remains roughly constant around 90 seconds. These results again highlight that minimizer-based filtering benefits from larger k-mer sizes, whereas exact indexed lookup of every k-mer does not.

The memory usage as a function of k remains moderate for all methods, but the trends are again distinct. K2Rmini stays almost constant around 7–8 MB across the full range of tested values. Deacon remains below 20 MB, with only a small increase as k grows. SBWT increases more noticeably, specifically after each power of two (32 and 64 in the plot), from about 7–8 MB at small k to about 20 MB for the largest tested values, especially on negative queries. BackToSequences has the highest memory usage throughout this experiment, steadily increasing from about 50 MB to more than 60 MB. Overall, these experiments confirm that K2Rmini is not only the fastest of the exact methods considered here, but also the most memory-efficient and the least sensitive to both thread count and k-mer size.

9.2.3. Real data

To complement the synthetic benchmarks, we compared BackToSequences and K2Rmini on several real datasets with heterogeneous sequence lengths and structures, summarized in Table 9.1. The evaluation includes Oxford Nanopore ultra-long reads (`SRR23365080`), PacBio HiFi reads of 10–20 Kbp (`SRX7897685-8`), and Illumina 250 bp reads (`ERR3239454`). We also evaluated assembled sequences from the Logan project [Chi+24], including both contigs and unitigs (`SRR7853572`), and the complete human T2T reference genome (`GCF_009914755.1`), which represents very long contiguous sequences. These experiments were run on a laptop equipped with a 13th Gen Intel Core i9-13950HX processor, 64 GB of RAM, and an SSD, using 8 threads. For the positive case, the queried patterns were sequences of length 1000 selected from each dataset, using 100 patterns for short reads and unitigs so as to total 10^6 queried bases. For the negative case, the queried patterns were random sequences of length 1000. Across all datasets and in both positive and negative settings, K2Rmini is consistently faster than BackToSequences in both CPU time and wall-clock time.

The largest gains are observed on ONT and HiFi reads. On ONT data, K2Rmini is 10.55×

faster than BackToSequences in CPU time and 4.94× faster in elapsed time for positive queries, while the speedup increases to 17.74× and 16.09× on negative queries. On HiFi reads, the gains are even larger for positive queries, reaching 26.94× in CPU time and 27.41× in elapsed time, and remain very high on negative queries with speedups of 17.23× and 16.55×. On Illumina reads, the improvement is more moderate but still substantial, ranging from 5.53× to 5.98× in CPU time and from 3.97× to 4.48× in elapsed time. Overall, these results confirm that the minimizer-based prefilter is especially effective on raw reads, with the strongest gains on long-read datasets.

The same trend holds on assembled and reference sequences, although with more variable gains. On the T2T genome, K2Rmini is 4.00× faster in CPU time and 4.19× faster in elapsed time for positive queries, while negative queries lead to much larger speedups of 15.19× and 11.10×. On Logan contigs, the gains are stable between positive and negative cases, around 7.3× to 7.4× in CPU time and 5.8× to 5.9× in elapsed time. Logan unitigs show the smallest difference, with speedups around 2.5× in CPU time and 1.9× in elapsed time in both settings. More generally, negative queries tend to benefit more from K2Rmini than positive ones, which is consistent with the minimizer filter rejecting most sequences early and therefore avoiding the exact counting phase. Overall, K2Rmini consistently outperforms BackToSequences across all tested datasets, with the strongest improvements on long reads and on negative queries.

Table 9.1.: Runtime comparison between K2Rmini and BTS using 8 threads. + indicates positive queries while – indicates negative ones.

Experiment	**CPU time (s)**			**Elapsed time (s)**		
	K2Rmini	BTS	Speedup	K2Rmini	BTS	Speedup
ONT +	1,850.65	19,529.04	×10.55	531.83	2,626.34	×4.94
ONT −	959.72	17,021.16	×17.74	125.13	2,013.64	×16.09
HiFi +	279.46	7,527.53	×26.94	33.11	907.56	×27.41
HiFi −	273.75	4,716.76	×17.23	33.70	557.88	×16.55
Illumina +	278.24	1,537.61	×5.53	43.56	173.02	×3.97
Illumina −	267.35	1,597.82	×5.98	40.41	180.89	×4.48
T2T +	55.85	223.67	×4.00	8.88	37.19	×4.19
T2T −	9.81	148.98	×15.19	2.31	25.63	×11.10
Contigs +	14.05	102.10	×7.27	2.18	12.66	×5.81
Contigs −	13.83	102.29	×7.40	2.15	12.70	×5.91
Unitigs +	76.56	191.45	×2.50	13.87	26.49	×1.91
Unitigs −	75.29	192.07	×2.55	14.07	26.65	×1.89

9.3. Limitations and perspectives

This work has demonstrated that a search strategy based on SIMD-accelerated minimizer filtering is fast and resource-efficient across a broad spectrum of applications. Similar tools are being developed for screening sequence repositories for antimicrobial resistance mutations, emerging pathogen surveillance, or sequencing contaminant filtration [CLC25]. Currently, the main limitation of our implementation is that the indexation of the patterns is single-threaded and relies on a generic hash table. One promising direction to improve the indexation step would be to implement a concurrent hash table that relies on minimizers for grouping keys, such as the one recently described in kache-hash [KPP26]. This would be especially relevant since we already have efficient methods to compute both minimizers and k-mer hashes, and we could thus easily batch insertions and queries. A second direction for improvement would be to parallelize the parsing step by attributing independent chunks of the input file to each thread, which would fully benefit from our vectorized parsing library and reduce the communication overhead.

Future work could explore alternative search strategies, including using sparser minimizer schemes to further reduce the number of minimizers to query [Pel+23; GP24; GLP25; ABP25; Gol+25; STO26]. We could also imagine a middleground between the costly exact search used in the second pass and doing no search at all at the cost of false positives, possibly by using small filters for faster queries with a lower false positive rate.

Finally, as the throughput of the core algorithm increases, performance bottlenecks shift to other components of the pipeline, mostly I/O bounded. While uncompressed FASTA/Q files are unnecessarily wasteful, reading input from compressed FASTA/Q files is a major bottleneck, highlighting the need for more performant alternatives [TD25; Pat+25].

First discussion

Ragnar Groot Koerkamp concluded his thesis [Gro25a] with the following observation:

> Provably optimal software consists of two parts: a provably optimal algorithm, and a provably optimal implementation of this algorithm, given the hardware constraints. This can only be achieved through algorithm/implementation co-design, where hardware capabilities influence design choices in the algorithm.

This view captures rather well what I tried to do throughout this part. Implementation is too often treated as a detail that comes after the algorithm is fixed, yet the chapters of this part show that the distinction is artificial: parsing, hashing, and minimizer selection all benefited from being redesigned with the hardware in mind. Doing so allowed us to reach the memory bandwidth for FASTA/Q parsing in Chapter 6, to compute ntHash at more than 2.5 Gbp/s in Chapter 7, and to extract minimizers at less than 2 ns/bp in Chapter 8. Stacking these building blocks together made it possible to filter sequences at 2 Gbp/s on a consumer laptop in Chapter 9, a regime that would have been out of reach by treating each layer in isolation.

A recurring observation across these contributions is that simplicity tends to win. Branchless code and regular access patterns turned out to be much easier to push close to the hardware limit than elaborate constructions with better asymptotic guarantees. This is not a universal rule, but it does sit somewhat at odds with the quest for lower complexity that drives parts of theoretical computer science. I would not argue against asymptotic improvements, of course, since they remain essential at very large scales, but the constants hidden behind complexity bounds deserve at least as much attention when the building blocks they describe run trillions of times in real pipelines.

While these results show the gains we can extract on current hardware, there are still many directions worth exploring. A natural one concerns the way we handle sequential dependencies. Both rolling hashes and minimizer selection are inherently sequential, and we worked around this by processing independent chunks of the input in parallel SIMD lanes. This strategy is effective, but it forces the input to be reloaded into multiple lanes, which can be expensive to fetch from memory for long sequences that no longer fit in cache. Designing a true single-stream variant that hides the dependency chain without splitting the input would reduce memory traffic and could matter especially in pipelines where the sequence has just been read from disk.

Another important direction concerns the theoretical foundations of the building blocks we use. As I noted in Chapter 7, the most efficient rolling hashes in the bioinformatics ecosystem have surprisingly few formal guarantees, and the biases I observed on ntHash are a concrete illustration of this gap. The community would benefit from rolling hashes that are both fast and provably well-behaved in terms of collisions and distribution, rather than choosing between the two. Ongoing work by Dufresne et al. [DGD24] on reversible hash functions tailored to DNA is a promising step in this direction, aiming at both "speed of encoding and decoding" and "dispersion of the hashed values across the integer space".

The hardware landscape has also shifted quite a bit during my PhD. The most visible change is the democratization of ARM in personal and server-grade machines, which made it essential to provide implementations that work well outside the x86 world. We made sure throughout this part that every contribution targets both x86 and ARM, which should keep the code relevant as the diversity of CPUs continues to grow. Maintaining several architecture-specific implementations remains a sizeable burden though, and I see it as a real obstacle to the broader

adoption of SIMD outside specialized libraries. Portable abstractions such as WebAssembly may eventually offer a unified target, but their vector support is still well behind native SIMD as of today. Closer to our needs, new languages such as Mojo[4] and domain-specific projects such as Shannon x Cray[5] tackle this challenge at the level of the compiler, by letting authors describe algorithms in a higher-level way that the compiler can then specialize to each target. Whether any of these efforts will scale to the kind of code we wrote here remains an open question, but this direction is one I find very exciting.

The most obvious omission in this part is the GPU. Modern GPUs offer an order of magnitude more parallelism than CPUs, and many bioinformatics workloads should in principle benefit from them. In practice, the main limitation we face is data movement: transferring sequences from main memory to GPU memory is often more expensive than the SIMD processing we would gain on the GPU itself. Processing-in-memory technologies promise to remove this bottleneck by performing computation close to where the data lives, but they remain quite niche and not particularly affordable. A more accessible avenue is the unified memory model adopted by recent architectures such as Apple Silicon, where the CPU and GPU share the same physical memory and can exchange data with near-zero cost. If this model becomes more widespread, I think it could finally make GPU acceleration practical for sequence processing pipelines, and provide a natural follow-up of the work presented in this part.

[4] https://mojolang.org/
[5] https://sxc.inria.fr/index.html

Part III.

Locality-preserving representations of k-mer sets

Keep your friends close, and your k-mers closer.

—not Sun Tzu

Part I established that careful implementation can push sequence processing close to the hardware limit. Now we ask a different question: can the representation of a k-mer set itself be a source of performance gains? The central observation is that consecutive k-mers extracted from a genomic sequence are not independent. A representation that ignores this locality pays for it in both memory and query time. Chapter 10 surveys existing approaches and establishes the metrics used throughout. Chapters 11 and 12 introduce CBL, a dynamic structure built on minimizer-partitioned sorted buckets that supports efficient set operations. Chapter 13 presents Brisk, which stores k-mers implicitly inside super-k-mers for a lower per-k-mer footprint. Chapter 14 pushes this idea further with hyper-k-mers, reducing the space overhead asymptotically to 4 bits / k-mer.

10. Background on *k*-mer sets

Spectrum-preserving string sets, introduced in Chapter 2, offer one compact way to represent a k-mer spectrum. By itself, however, such a representation gives no direct way to test whether an arbitrary k-mer belongs to the spectrum or not. Building an efficient *index* is therefore a necessary step in any k-mer-based pipeline, from read mapping and taxonomic classification to genome assembly. Depending on the application, this index may be built from an SPSS (repetitive or not), directly from raw reads, or from a reference genome, all of which may contain repeated k-mers.

At the most basic level, a k-mer *set* supports a single operation: *membership*, testing whether a given k-mer is present or not. Many applications require more than this and associate each k-mer with satellite information such as an abundance count or a color. A k-mer *dictionary* extends a k-mer set with a *lookup* operation: given a k-mer, it returns a unique integer identifier that serves as a key into any external array of satellite values, or -1 if the k-mer is absent. Both sets and dictionaries are commonly used in a *streaming* regime, where queries are issued for every k-mer extracted consecutively from a sequence. This modality is prevalent enough that several structures are explicitly designed for it, favoring cache-efficient access over runs of consecutive k-mers rather than minimizing the cost of individual queries.

This chapter focuses on three approaches representative of the state of the art for exact k-mer set indexing, and a broader survey of the landscape can be found in [Mar24]. SSHash [Pib22; PP26] uses compact hashing over the super-k-mer structure of the input to achieve very fast queries at moderate space. The SBWT [APV23] exploits the colexicographic ordering of k-mers in the de Bruijn graph to compress the index close to the navigational lower bound, at the cost of slower $\mathcal{O}(k)$-time queries. FMSI [SVB23; SVB24; SVB25] indexes a masked superstring with an FM-index, offering a different point on the space-time tradeoff.

10.1. Path locality and hashing

> **Different notations**
>
> We reuse the notations from Section 4.3: the minimizer size is denoted m while k denotes the number of bases in a window, i.e. $k = w + m - 1$.

A first family of k-mer indexes exploits the *path locality* of consecutive k-mers in the input. As established in Section 4.3, k-mers at successive positions tend to share the same minimizer and can therefore be grouped into super-k-mers. In the context of indexing, this grouping has two important consequences. First, there are far fewer super-k-mers than k-mers: each super-k-mer spans on average $(w+1)/2$ k-mers, the inverse of the minimizer density, reducing the number of objects to be indexed by the same factor. Second, consecutive k-mers within a super-k-mer are stored contiguously, so a streaming query traversing a sequence left to right will access the same super-k-mer repeatedly before moving to the next, yielding favorable cache behavior. More broadly, the super-k-mer structure induces a partial order on k-mers that an index can preserve: k-mers within the same super-k-mer can be assigned consecutive identifiers, enabling fast satellite data access for streaming queries, a property formalized and exploited by LPHash [PSL23].

10.1.1. Sparse and skew hashing: SSHash

A *minimal perfect hash function* (MPHF) for a set $\mathcal{X}$ of n keys is a bijection $f : \mathcal{X} \to [0, n)$ mapping each key to a distinct integer with no gaps [Leh+26]. MPHFs can be constructed using close to $\log_2 e \approx 1.44$ bits per key in expectation [LRCP17; PT21; Gro25b], far below the cost of storing the keys themselves. Since f assigns an arbitrary value in $[0, n)$ to any input outside $\mathcal{X}$, a structure relying on an MPHF must store the keys or compact fingerprints separately to verify membership.

SSHash [Pib22; PP26] takes an SPSS as input, identifies super-k-mer boundaries on the fly to record their offsets within the strings, and builds an MPHF over the set of minimizers to index those offsets, while retaining the SPSS strings as the underlying storage. Rather than hashing k-mers directly, the MPHF operates on the much smaller set of minimizers. This is the *sparse* component of the design: since each super-k-mer spans on average $(w + 1)/2$ k-mers, the MPHF is built over a key set far smaller than the k-mer set. The input strings are stored at 2 bits per base, alongside a prefix-sum array encoding partition sizes and an offsets array pointing to super-k-mer positions within the strings. Importantly, the SPSS strings remain stored as a single contiguous block: minimizer partitions are virtual, materialized through the auxiliary arrays rather than by physically grouping super-k-mers in memory. Figure 10.1 illustrates the resulting data structure layout.

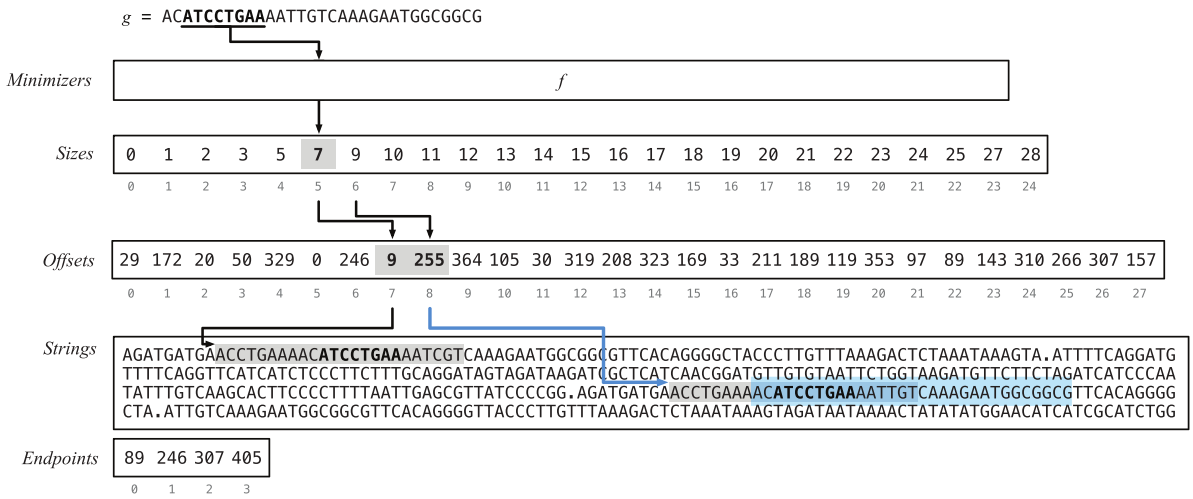


Figure 10.1.: Overview of the SSHash dictionary data structure, from Pibiri [Pib22]. The input strings are stored contiguously; minimizers are mapped to partition identifiers via an MPHF; the Sizes and Offsets arrays locate each super-k-mer within the strings.

The *skew* component addresses the non-uniformity of partition sizes: even with a random hash function, most minimizers appear only once while a few minimizers appears in a disproportionately large number of super-k-mers, yielding highly skewed minimizer partition sizes. In particular, for $k = 31$ and $m = 20$, over 97% of minimizers have a single occurrence [Pib22, table 1], while the most frequent minimizer appears more than 36000 times in the SPSS of a human genome. This observation was later confirmed by combinatorial analyses [IMS24; ILMS26].

SSHash encodes the prefix-sum of partition sizes using Elias-Fano [Eli74; Fan71], which compresses increasing integer sequences close to their information-theoretic minimum. A secondary *skew index* groups frequent minimizers by partition size and builds a separate MPHF per group, limiting the number of super-k-mers to inspect at query time.

To answer a lookup query for k-mer x, SSHash computes the minimizer of x, retrieves the corresponding partition via the MPHF, and scans its super-k-mers until x is found, returning its identifier or -1 if it is absent. Streaming queries benefit from caching the last minimizer: consecutive k-mers sharing the same minimizer extend the previous match without recomputing

the partition, reducing the average number of memory accesses per query. In practice, SSHash achieves around 7–10 bits / k-mer and answers individual queries in hundreds of nanoseconds [Pib22, table 3].

10.1.2. Improvements on SSHash

A subsequent refinement [PP26] improves query efficiency on two fronts. First, rather than storing the start of each super-k-mer and scanning it, the refined index stores a single position per minimizer occurrence; combined with the offset of the minimizer within the query k-mer x, this is sufficient to recover the exact location of x in the SPSS in constant time. Second, a *tag* array classifies each minimizer as singleton, light, or heavy based on how many super-k-mers share it; for *singleton* minimizers, the unique position is encoded directly in the tag itself, so that a lookup requires only two cache misses: one to evaluate the MPHF and one to read the tag. Since over 97% of minimizers are singletons for $m \geq 20$, this shortcut applies to the vast majority of queries. Figure 10.2 illustrates the two layouts side by side.

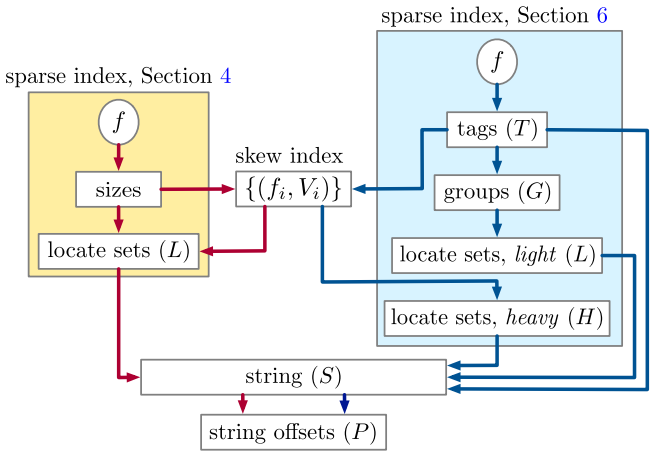


Figure 10.2.: Comparison of the original and refined SSHash data structures, from Pibiri and Patro [PP26]. The tag array classifies minimizers as singleton, light, or heavy; singleton minimizers have their position encoded directly in the tag, avoiding any further memory access.

LPHash [PSL23] takes a slightly different approach: rather than building a complete dictionary with the SPSS strings as underlying storage, it constructs a locality-preserving MPHF alone, without retaining the strings. This omission means that a k-mer absent from the indexed set will also receive some identifier in $[0, n)$ rather than -1, so membership cannot be verified without external storage. In exchange, the space footprint is substantially smaller, achieving 0.6–0.9 bits / k-mer, and lookup queries are an order of magnitude faster than in SSHash.

10.2. Colexicographic clustering and spectral BWT

10.2.1. Colexicographic order and Wheeler graphs

A second family of k-mer indexes stems from sorting k-mers in *colexicographic* order, comparing them from right to left. Under this ordering, k-mers sharing the same length-$(k-1)$ suffix are adjacent, since they only differ in their leftmost character. Recall that in a node-centric de Bruijn graph, an edge from k-mer x to k-mer y exists iff the last $k-1$ characters of x match the first $k-1$ characters of y; we label such an edge by the last character of y, i.e. the character appended when extending x into y. The colexicographic ordering of k-mers makes the de Bruijn

graph a *Wheeler graph* [GMS17]: nodes can be totally ordered such that edges with the same label preserve this ordering. A key consequence is that for any character c, c-labeled edges map a contiguous range of sources to a contiguous range of destinations, and vice versa. This property enables *backward search*: membership of a query k-mer x can be tested by starting from the interval of all k-mers ending in the last character of x, then iteratively narrowing it by prepending characters of x from right to left, each step intersecting with a contiguous range. Crucially, the colexicographic rank of a k-mer implicitly encodes its identity, so the structure does not require explicit storage of the $2k$-bit label of each k-mer, only small per-node annotations. This is the root of the space efficiency of BWT-based approaches. The BOSS structure [BOSS12] realizes these ideas for the edge-centric de Bruijn graph; the SBWT [APV23], described below, extends them to the node-centric view.

10.2.2. The spectral BWT and subset rank

The spectral Burrows-Wheeler transform, or SBWT [APV23], builds directly on the colexicographic structure. Given a k-mer set S, the k-mers are sorted colexicographically and grouped by their length-$(k-1)$ suffix. For each group, only the first k-mer in colexicographic order carries a non-empty set of characters: those that label outgoing edges from that group in the node-centric de Bruijn graph. Source k-mers (those with no predecessor in S) are augmented with dollar-padded prefix k-mers to give them a virtual predecessor from a dummy root. When a k-mer has multiple predecessors, only the colexicographically smallest is retained, so that every k-mer has exactly one incoming edge in the resulting structure. The resulting sequence of character subsets, one entry per k-mer, is the SBWT. Figure 10.3 illustrates the construction on a small example.

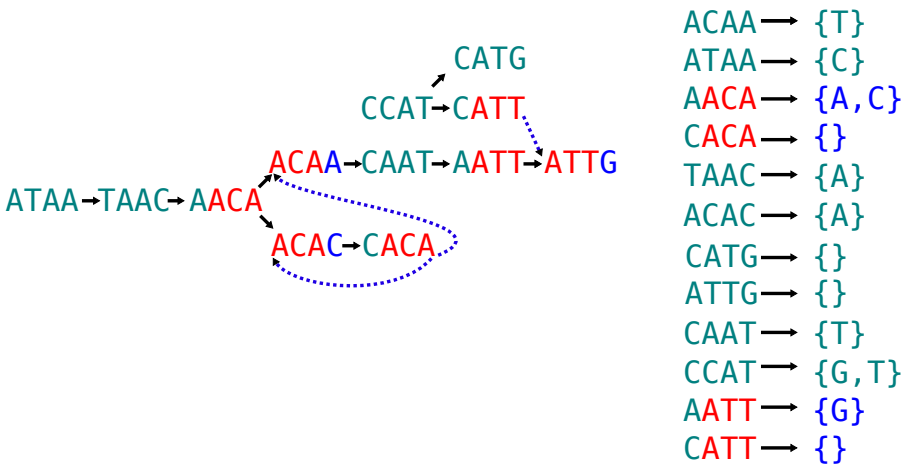


Figure 10.3.: Example of SBWT construction on a small k-mer set, courtesy of Camille Marchet. The k-mers are sorted colexicographically, as listed on the right. Only the first k-mer in each length-$(k-1)$ suffix group (highlighted in red) carries the outgoing edge subset, while the other edges (dashed in blue) are pruned. After construction, the SBWT only retains the subsets of outgoing edge labels, listed on the right. Note that this figure omits the dollar-padded prefix k-mers for clarity.

Membership queries rely on a single primitive: *subset rank*. Given a character c and a position i, $\text{SubsetRank}_c(i)$ counts how many of the first i subsets contain c. To query k-mer $x = x_1 \dots x_k$, backward search starts from the full interval $[1, n]$ and iterates over the characters of x from left to right, narrowing the interval at each step using:

$$\ell' = 1 + C[c] + \text{SubsetRank}_c(\ell - 1) + 1, \quad r' = 1 + C[c] + \text{SubsetRank}_c(r)$$

where $C[c]$ counts k-mers whose last character is smaller than c. After k steps, a non-empty interval indicates that x is present, and the resulting ℓ is the colexicographic rank of x, serving directly as its integer identifier. The total cost is $\mathcal{O}(k)$ subset rank operations. Note that, using the same process, the SBWT can actually query any m-mer for $m \leq k$.

A further benefit is that de Bruijn graph navigation comes for free: successors and predecessors of any k-mer can be read off from the index with a single step of the same algorithm, without auxiliary structures.

Several data structures can be used to support subset rank queries, offering different time-space tradeoffs. The most commonly used in practice is the *bit-matrix* SBWT, which stores one bitvector of length n per character $c \in \Sigma$, where bit i is set if $c \in X_i$. $\text{SubsetRank}_c(i)$ then reduces to a standard rank query on the c-th bitvector, and occupies around 4–5 bits / k-mer[1] [APV23]. More space-efficient variants exist, the most compact being the *subset wavelet tree* SBWT, which compresses the subset sequence to its zeroth-order entropy, achieving 2.6–2.8 bits / k-mer in practice at the cost of slower queries.

The SBWT can be further augmented with a *longest common suffix* array to accelerate streaming queries [ABP23], by allowing consecutive k-mer lookups to reuse intervals computed at the previous position. This same array also enables the SBWT to compute *matching statistics* between a query sequence and the indexed k-mer set [MABP25], extending its use to pseudo-alignment and sequence similarity tasks.

10.3. Masked superstrings

> **Note**
>
> As this approach was published recently, it was not included in our comparisons in the following chapters.

Masked superstrings [SVB23] generalize spectrum-preserving string sets by allowing the representation to include k-mers absent from the input, saving space at the cost of introducing spurious k-mers. A bitmask is added to resolve the resulting ambiguity: a *masked superstring* is a pair (S, M) where S is a single string containing all k-mers of the indexed set K as substrings, and M is a binary mask indicating which k-mer occurrences in S are *active*, i.e. belong to K. Conceptually, this replaces the de Bruijn graph with an overlap graph admitting edges of any length, providing better compression power when the k-mer set does not follow the spectrum-like property, as in subsampled or high-diversity metagenomic data.

Several mask variants serve different purposes. A *max-one* mask, maximizing active k-mers, optimizes for membership queries. A *min-one* mask, activating each k-mer exactly once, enables dictionary queries by giving each k-mer a unique identifier. A *min-runs* mask minimizes the number of runs of 1's, making the mask itself more compressible.

While computing the shortest masked superstring is NP-hard, approximately shortest masked superstrings can be computed from an input k-mer set or a precomputed SPSS using the greedy tool KmerCamel [SVB23].

FMSI [SVB25] indexes the masked superstring via the *Masked Burrows-Wheeler Transform* (MBWT), an extension of the classical BWT that incorporates the mask. The MBWT is computed by pairing each superstring symbol with its mask bit, sorting pairs lexicographically on the superstring symbol alone, then unzipping the result into a transformed string S' and transformed mask M'. Membership of a query k-mer x is answered by backward search in S' to find a position range $[i, j]$, then checking M' for any active k-mer in that range; the total cost is

[1] The bit-matrix itself uses exactly 4 bits / k-mer; the overhead above this comes from rank support and the dollar-padded prefix k-mers added for source nodes.

$\mathcal{O}(k)$ rank operations. In practice, FMSI achieves around 3–4 bits / k-mer [SVB25], at the cost of slower queries compared to SSHash or SBWT. Unlike SPSS-based approaches, it does not rely on the spectrum-like property, so its space cost is less sensitive to subsampled or high-diversity k-mer sets.

FMSI can be extended to support k-mer set operations and dynamic updates via *f-masked superstrings* [SVB24]. The binary mask is replaced by an integer vector so that k-mer occurrences from different input sets can be distinguished, and a *demasking function* f specifies how to combine the integer values seen at all occurrences of a k-mer to decide its membership in the final set. Union, intersection, difference, and symmetric difference of two indexed k-mer sets are then implemented by concatenating their masked superstrings, reapplying the appropriate demasking function to compute the resulting mask, and optionally compacting the structure to reduce its size. Individual k-mer insertions and deletions follow as special cases of these set operations with singleton sets, providing dynamic updates without rebuilding the index.

10.4. Comparison

The three approaches sit on a clear space-time tradeoff curve. SSHash is the fastest by a wide margin (74 ns/k-mer for streaming), at a moderate space cost of 10.01 bits / k-mer; it benefits from the SPSS strings being kept as contiguous raw storage, enabling cache-friendly lookups but inflating the bit budget. The SBWT natively uses 4–5 bits / k-mer, doubled to 10.50 bits / k-mer in this benchmark setting where both strands of each k-mer are indexed for canonical queries; with 266 ns/k-mer for streaming, it occupies a middle ground in time and offers de Bruijn graph navigation and matching statistics out of the box. FMSI is by far the most compact at 3.31 bits / k-mer, but its query time (1176 ns/k-mer) is significantly slower than SSHash, since each query requires backward search over the FM-index without the cache benefits of super-k-mer partitioning.

The space gap between FMSI and the others widens further for k-mer sets that do not follow the spectrum-like property (subsampled sketches, high-diversity metagenomes), where SPSS-based representations become fragmented. Conversely, SSHash's advantage is most pronounced for streaming queries on long sequences, where super-k-mer caching amortizes lookups across consecutive k-mers. Among the three, only FMSI extends natively to dynamic updates and set operations through f-masked superstrings; SSHash and SBWT remain static and rely on external mechanisms for these tasks, the subject of dedicated chapters later in this thesis.

Table 10.1 summarizes these properties, using values from the benchmark of Pibiri and Patro [PP26] on the human genome at $k = 31$, publicly available at https://github.com/jermp/kmer_sets_benchmark.

Table 10.1.: Summary of the three k-mer indexing approaches discussed in this chapter, with space and streaming lookup values for the human genome at $k = 31$ from the benchmark of Pibiri and Patro [PP26].

Approach	Input	Static / dynamic	Space (bits/k-mer)	Streaming lookup (ns/k-mer)
SSHash [PP26]	SPSS	static	10.01	74
SBWT [APV23]	SPSS	static	10.50[2]	266
FMSI [SVB25]	Masked superstring	static or dynamic[3]	3.31	1176

[2] The 4–5 bits / k-mer reported earlier in the chapter doubles here because both strands of each k-mer are indexed to support canonical queries.
[3] Dynamic updates via *f-masked superstrings*.

11. Necklaces and minimizers

> **Note**
>
> This chapter is partially adapted from Martayan et al. (2024), accepted to ISMB 2024 and published in Bioinformatics.

While we have seen in Chapter 4 that minimizers can be used as a way to partition k-mers while preserving locality, we present in this chapter a new approach based on smallest cyclic rotations, or *necklaces*. We will discuss some of their properties, highlight their connection to lexicographic minimizers and explain how they can represent k-mer sets efficiently. In particular, one property that will prove particularly useful is that necklaces of consecutive k-mers often share long prefixes.

11.1. Necklaces and their properties

Definition 11.1 (Necklace)**.** We define the *necklace* $\langle x \rangle$ of a string x as its smallest cyclic rotation:

$$\langle x \rangle = \min_i \mathrm{rol}^i(x)$$

It's pretty clear that the necklace transformation is not injective, for instance `CAT`, `ATC` and `TCA` all have `ATC` as a necklace. Intuitively, each necklace of size k acts as a representative of its k cyclic rotations, as depicted in Figure 11.1. The following property confirms this intuition and shows that necklaces of size k roughly account for a fraction $\frac{1}{k}$ of all k-mers:

Proposition 11.1 (Number of necklaces [Lot97])**.** Given an alphabet of size σ, the number of necklaces of size k is given by

$$N(k) = \frac{1}{k} \sum_{d|k} \varphi\left(\frac{k}{d}\right) \sigma^d \sim \frac{\sigma^k}{k}$$

where φ denotes Euler's totient function.

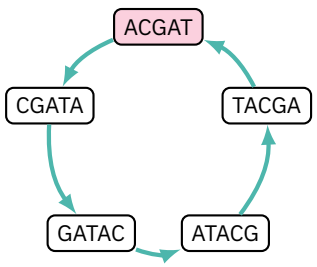


Figure 11.1.: Example of all cyclic rotations of `GATAC`, with the necklace `ACGAT` highlighted in color.

11.1.1. Ranking necklaces

Because they only represent a fraction $\frac{1}{k}$ of the universe, it should be possible to have an encoding of necklaces that saves about $\lg k$ bits. One way to obtain such encoding is to *rank* necklaces: given a necklace $\langle x \rangle$, find i such that $\langle x \rangle$ is the i-th smallest necklace of size k. Figure 11.2 illustrates the effect of ranking on necklaces of size 4.

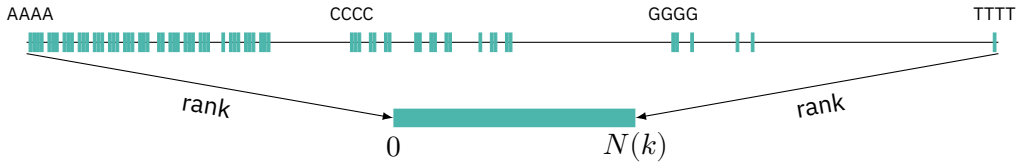


Figure 11.2.: Repartition of necklaces in Σ^4, applying rank maps all the necklaces to $[0, N(k))$.

Sawada and Williams [SW17] proposed an algorithm to compute the rank of a necklace of size k in $\mathcal{O}(k^2)$, which is still the best known as of today. Unfortunately, this running time is quite prohibitive for practical applications on large collections of necklaces where we typically aim for near-constant queries. Finding a faster way to compute, even approximately[1], the rank of a necklace is still an open question.

11.1.2. Bijective encoding of *k*-mers

We've seen that the necklace transformation is lossy in the sense that we cannot tell which of its cyclic rotations it represents. Therefore, in addition to the necklace, we want to keep track of the rotation offset in order to make the transformation reversible.

Definition 11.2 (Rotation offset)**.** We define the *rotation offset* $\rho(x)$ of a k-mer x as the minimum number of left rotations required to obtain its necklace:

$$\rho(x) = \arg\min_i \operatorname{rol}^i(x) \qquad \operatorname{rol}^{\rho(x)}(x) = \langle x \rangle$$

Since there are k such rotation offsets, it requires $\lceil \lg k \rceil$ bits to encode $\rho(x)$. Thus, when combined with the rank encoding of the necklace, this yields a fully reversible transformation whose total size matches the original k-mer up to an extra rounding bit.

11.1.3. Encoding canonical *k*-mers

The encoding discussed above does not take the canonical aspect into account, but we can adapt the technique of Wittler [Wit23] to save one additional bit when k is odd. The idea is that, when using the encoding presented in Section 6.2, the complement operation flips exactly one bit for each base (e.g. `A = 00` $\leftrightarrow$ `T = 10`). Therefore, when k is odd, the parity of the number of 1s (efficiently computed with a `popcount`) changes between a k-mer and its reverse-complement. We can thus define a k-mer as canonical if its number of 1s is odd, and safely erase its last bit: if the remaining $2k-1$ bits have an odd number of 1s we know that the last bit must be 0, otherwise it must be 1. From this $(2k-1)$-bit encoding, we then apply the reversible necklace transformation, this time using a binary alphabet. An example of this canonical encoding is shown in Figure 11.3.

11.2. Necklaces of consecutive *k*-mers

We've seen how to encode individual k-mers as necklaces, but in most applications we actually want to index consecutive k-mers extracted from a sequence. We now discuss the benefits of using necklaces in this context, and how this can be used to design a new data structure.

[1] For instance, we could afford mapping the values to an output space that's 1% bigger than $N(k)$.

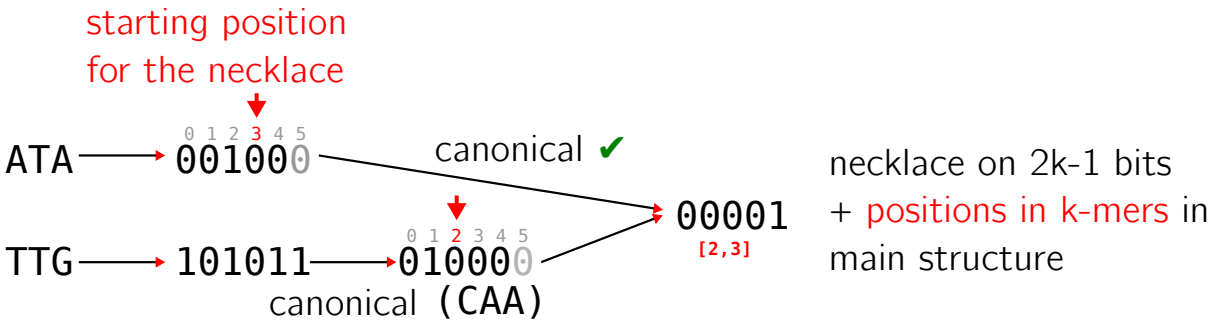


Figure 11.3.: Example of canonical necklace encoding for `ATA` and `TTG`. `TTG` has an even number of 1s so it is first transformed into its reverse-complement `CAA` before applying the necklace transformation. Both k-mers end up having the same binary necklace, but the rotation offsets distinguish them.

11.2.1. Runs of common prefixes

A key property of necklaces is that, when considering two consecutive k-mers x and y, their necklaces tend to share a long prefix. The intuition behind this is quite simple: if $x_1 = y_k$ their necklaces are completely identical, and if $x_1 \neq y_k$ they are likely to differ by a single letter at position $k - \rho(x)$, with $\rho(y) = \rho(x) - 1$. The only situation where the necklaces diverge is when the incoming letter y_k creates a rotation that is lexicographically smaller than the current necklace. This is more likely to happen when $\rho(x)$ is small, since the substitution then falls closer to the start of the necklace. This phenomenon is illustrated in Figure 11.4, where we can see runs of increasingly long prefix matches of length $k - \rho(x)$ as we advance. We can also notice that roughly 1/4 of the matches have length k, corresponding to the event where $x_1 = y_k$.

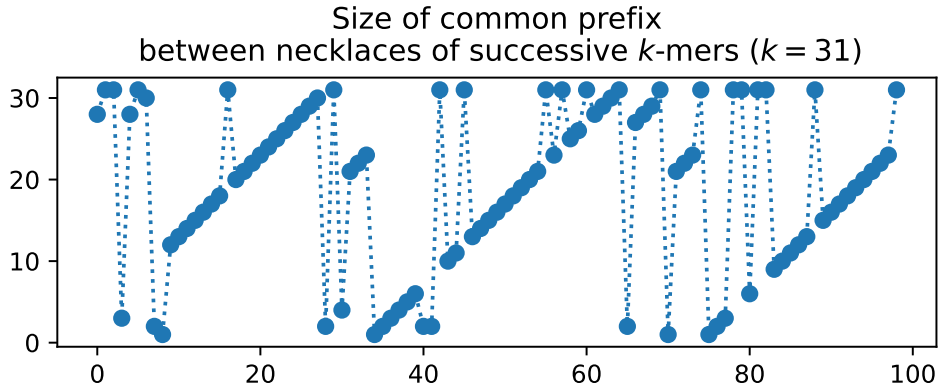


Figure 11.4.: Length of the common prefix between necklaces of consecutive k-mers with respect to their position in the sequence, simulated on a random sequence with $k = 31$.

11.2.2. Computing necklaces with minimizers

While computing a necklace individually requires $\mathcal{O}(k)$ operations since we have to consider every rotation, we can accelerate this process down to $\mathcal{O}(\log k)$ operations in the context of consecutive k-mers. The main property that we rely on is that the prefix of size m of a given necklace is always the smallest m-mer in the circular word. In particular, this m-mer is either fully included in the original k-mer or overlaps the k-mer's start and end, in that case we refer to it as a *boundary substring*. Boundary substrings change for each new k-mer, so we have to recompute them every time. Non-boundary substrings, however, are preserved for consecutive

k-mers. Because of this, we can use classical algorithms for minimizers (such as those discussed in Chapter 8) to compute the smallest non-boundary m-mer in $\mathcal{O}(1)$ amortized time. Moreover, we know that if m is large enough ($\Omega(\log k)$) the smallest m-mer is unique with high probability [ZKM20, lemma 9]. Therefore, by choosing $m = \Theta(\log k)$, we only have one non-boundary m-mer to consider w.h.p. and $m-1$ boundary m-mers to compute, leading to $\mathcal{O}(\log k)$ time overall. In practice, we found that using substrings of $m = 9$ bits gave the best results for k ranging from 31 to 63. Our implementation of this algorithm can compute 100M necklaces per second on a laptop with a M1 processor.

11.2.3. Super-necklaces

The connection between necklaces and lexicographic minimizers does not end here: just like minimizers can group consecutive k-mers into super-k-mers, consecutive necklaces can be grouped into *super-necklaces*. The idea is simple: as long as $\rho(y) = \rho(x) - 1$, i.e. as long as we are in the same run, only one letter is replaced at a time and we know its position. Therefore, assuming a run of r k-mers $x^{(1)}, \dots, x^{(r)}$ forming a sequence s, we can compactly encode all their necklaces $\left\langle x^{(1)} \right\rangle, \dots, \left\langle x^{(r)} \right\rangle$ as a tuple of strings containing $|s| = r + k - 1$ bases:

$$(u, v) = \left(s_{[\rho(x^{(1)})+1,\ |s|]}, s_{[1,\ \rho(x^{(1)})]} \right)$$

Using this representation, we can recover all individual necklaces by combining a prefix of the first string with a suffix of the second:

$$\left\langle x^{(i)} \right\rangle = \mathrm{pref}_{k-\rho(x^{(1)})+i-1}(u) \cdot \mathrm{suff}_{\rho(x^{(1)})-i+1}(v)$$

11.3. CBL: a dynamic data structure for *k*-mer sets

Building on the necklace representation described above, we now present CBL (Conway-Bromage-Lyndon), a compressed and dynamic data structure for exact k-mer set representation, available at https://github.com/imartayan/CBL. As highlighted in the previous section, necklaces of consecutive k-mers tend to share long common prefixes, making them amenable to prefix-based compression. Moreover, since k-mer sets extracted from biological sequences are sparse in $[0, 2^{2k-1} - 1]$, necklaces concentrate towards small values and many share common prefixes. We exploit this skewness with a quotienting strategy, storing prefixes and suffixes separately. Figure 11.5 gives a schematic overview of the resulting structure.

11.3.1. Quotienting sets of necklaces

To achieve compression, we select a prefix size p and store prefixes and suffixes of necklaces separately, the suffix being $2k - 1 - p$ bits long. The rationale is that many necklaces share the same prefix, particularly since their distribution is skewed towards smaller values. Quotienting thus concentrates the bulk of the set into a small number of prefix buckets, each holding a short list of suffixes.

11.3.2. Prefix data structure

For the storage of prefixes, we use a bitvector of size 2^p. To facilitate rapid access and insertion by rank, a structure supporting efficient dynamic rank/select operations is required. Several such structures have been discussed in the literature [Vig08; GP13; ZAK13], but dynamic variants are comparatively rare. We adopt an implementation based on [MV20; PK21; DPR22] that supports fast dynamic rank/select.

The key ingredient is a Fenwick tree, an efficient data structure for prefix sums of a dynamic array. Here the array elements are bits, so the binary rank at any position equals the prefix sum

up to that position. This arrangement allows rank to be computed in $\mathcal{O}(\log 2^p) = \mathcal{O}(p)$ time by traversing the tree, and updates are equally efficient.

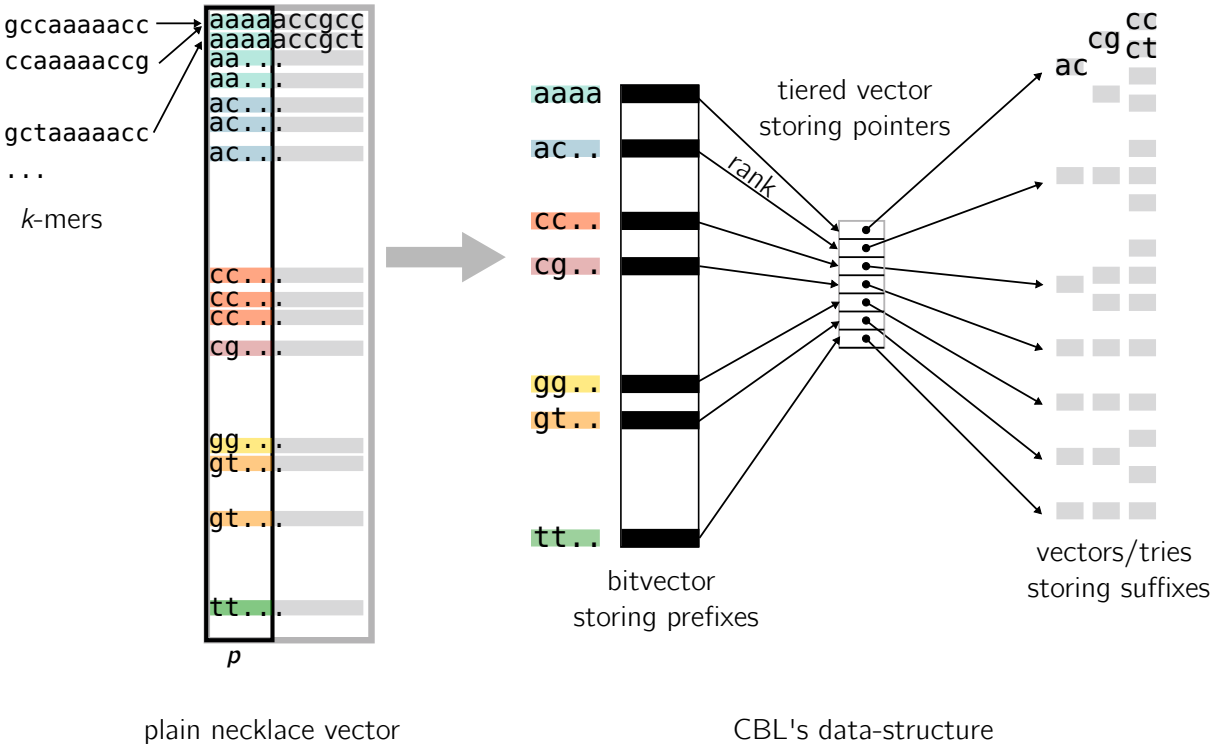


Figure 11.5.: Schematic view of CBL's data structure. For clarity, the example uses lexicographic rather than binary necklaces. The full necklace vector is shown on the left, with two overlapping k-mers sharing closely related necklaces. Prefixes are stored in a bitvector and associated to suffix buckets via their rank.

11.3.3. Associating prefixes to suffixes

For associating each prefix with its corresponding list of suffixes we use a tiered vector [BCEG17]. This structure maps each prefix rank to a bucket of suffixes by storing a pointer at the index corresponding to that rank. Insertions and deletions are supported at any position in $\mathcal{O}(n^\varepsilon)$ time (where n is the total number of elements and $\varepsilon < 1$), while access remains $\mathcal{O}(1)$.

Tiered vectors maintain a dynamic sorted array in the leaves of a shallow tree (internal depth 4 in our implementation). Fast insertion is achieved by filling available slots in the leaves; since this may disturb leaf order, internal nodes store the rotations needed to restore the correct ordering.

11.3.4. Suffix storage

As noted in previous studies, the distribution of small-sized genomic words is highly skewed. We therefore expect most buckets to hold only one or a few suffixes, with a minority containing many. To handle both cases efficiently, we use an adaptive strategy. Small buckets store suffixes in a contiguous vector that is scanned linearly. Large buckets (more than 1024 elements by default) switch to a trie, exploiting the high suffix similarity typically found there, e.g. from repetitive or low-complexity regions. The trie uses 1 byte per level; for vector-based storage, suffixes are likewise divided into bytes to minimize footprint. The rotation offset (see Definition 11.2) is encoded in the trailing bits of each suffix to make the transformation reversible.

11.3.5. Operations

The main operations (membership, insertion, deletion) follow the same pipeline:

1. compute the necklace and rotation offset of the k-mer (as described above), and append the rotation offset at the end of the necklace
2. split the result into a prefix q of p bits and a suffix r
3. look up the prefix in the bitvector
4. compute the rank of q using the Fenwick tree
5. retrieve the bucket associated to that rank via the tiered vector
6. perform membership/insertion/deletion of r in the bucket:
    - for small buckets, scan the vector linearly
    - for large buckets, navigate the trie byte by byte

When the necklaces of consecutive k-mers share the same prefix q, steps 1–5 only need to be performed once and the remaining operations are batched on the same bucket. This optimization is particularly effective when streaming all k-mers of a sequence.

11.3.6. Informal comparison to Elias-Fano

CBL's quotienting layout is analogous to Elias-Fano encoding for sorted integers: both split elements into high (prefix) and low (suffix) parts stored separately. The key difference is in how suffixes are handled. Elias-Fano omits an explicit prefix-to-suffix mapping and stores all suffixes contiguously, achieving higher compression but making insertion expensive by shifting many elements. Dynamic adaptations of Elias-Fano have been proposed by Pibiri and Venturini [PV17] but have no practical implementation yet. CBL trades some compression for full dynamicity: every layer of the structure supports insertions and deletions efficiently.

11.3.7. Implementation details

The necklace and rotation offset are packed into a single primitive integer of up to 128 bits. The combined size $2k - 1 + \lceil \log_2(2k-1) \rceil$ must therefore stay below 128 bits, limiting support to $k \leq 59$.

The prefix size p is a compile-time parameter, supported up to 32 bits. We use $p = 24$ bits as a default, balancing the size of the prefix bitvector (2^p bits) against the load on suffix buckets. Increasing p for very large sets reduces bucket sizes and speeds up bucket operations.

11.4. Comparison against other indexes

We conducted extensive benchmarks on the CBL library, focusing on its ability to represent and manipulate k-mers in various biological datasets. All experiments were performed on a single cluster node with an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 128 GB of RAM, and Ubuntu 22.04. Experiment scripts, competing tools, and plot generation code are available at https://github.com/imartayan/CBL_experiments.

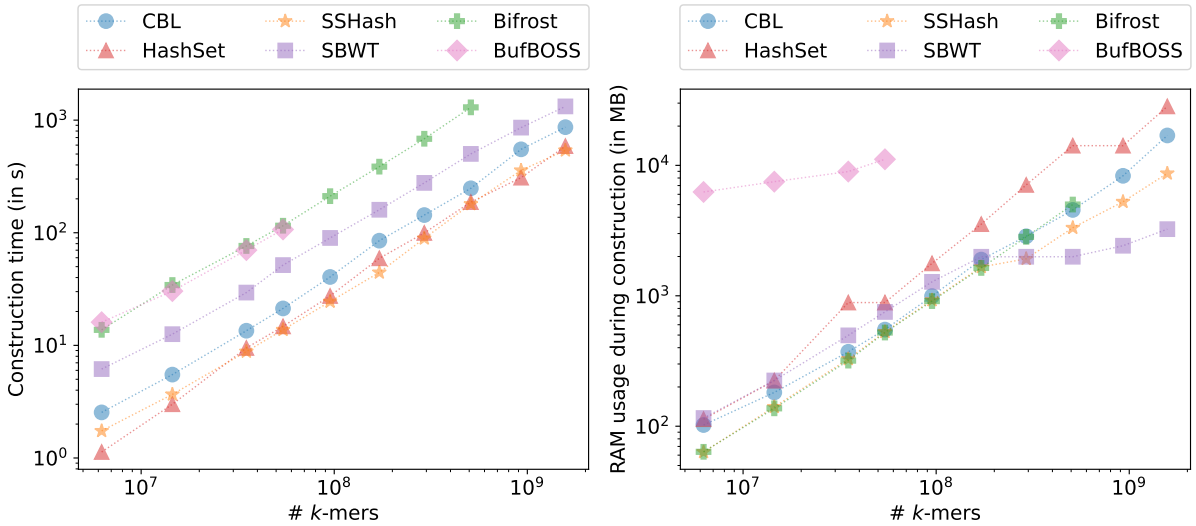
We compare CBL against two state-of-the-art static structures: SSHash [Pib22], a minimal-perfect-hash-based k-mer index, and SBWT [APV23], a succinct full-text k-mer index. Among dynamic structures we include Bifrost [HM20], a colored de Bruijn graph supporting insertions, and BufBOSS [Ala+21], a BWT-based de Bruijn graph with insertion/deletion buffers. We also include Rust's HashSet (based on Google's Swiss Tables) as a generic dynamic baseline.

11.4.1. Index construction

For our initial experiment, we built indexes on an expanding collection of bacterial genomes. We randomly selected subsets of genomes from which we computed a compacted de Bruijn graph (unitig set), doubling the subset size each time. The largest file comprised 1024 genomes and 1,574,701,184 distinct k-mers. Time and memory usage are reported in Figure 11.6.

SSHash is the best overall performer in terms of construction speed and memory, but it is static and requires duplicate-free input. HashSet is similarly fast but uses more memory. CBL is slightly slower than both while matching SSHash's memory footprint. SBWT is slower than these three despite being static, with memory comparable to HashSet. Bifrost is significantly slower but equally memory-efficient to SSHash and CBL. BufBOSS is the slowest and most memory-intensive tool.

The fact that CBL matches the throughput of a high-performance hash table while being equally memory-efficient is particularly noteworthy. Importantly, Bifrost, HashSet, and CBL are the only tools that accept any FASTA input directly. The other tools require a k-mer counting preprocessing step (e.g. KMC3 for SBWT and BufBOSS, unitig assembly for SSHash), which externalizes part of their construction cost and may impose heavy disk usage.



(a) Construction time.

(b) RAM usage.

Figure 11.6.: Time and RAM used when constructing various indexes on a growing bacterial genome collection from RefSeq, for $k = 31$ and $p = 28$ bits.

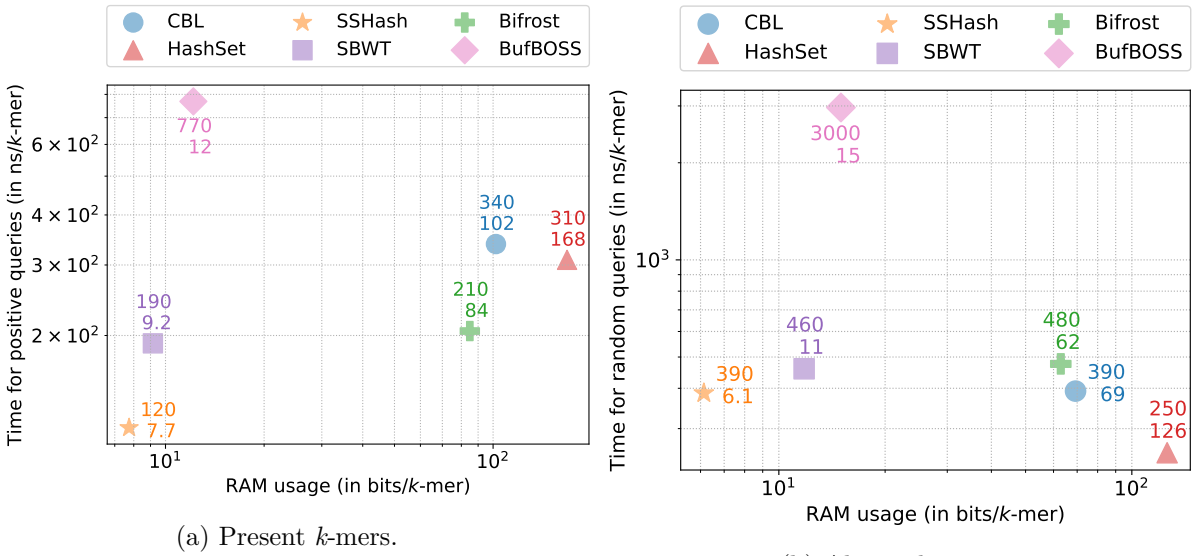
11.4.2. Queries

To evaluate query performance we benchmark positive queries (k-mers present in the index) and negative queries separately. Positive queries use the input unitigs' k-mers; negative queries use randomly generated k-mers with a negligible probability of being present. Results are shown in Figure 11.7, reporting query time and RAM usage per k-mer, averaged over a series of experiments.

As expected, static structures and BufBOSS use very little RAM during queries, while other dynamic structures are more memory-intensive. SSHash and SBWT are the fastest for queries; BufBOSS is the slowest. Among dynamic structures, CBL and Bifrost have the smallest memory footprint, while HashSet is the most demanding.

The relative ranking changes between positive and negative queries: Bifrost is fastest for positive queries (with CBL and HashSet close behind) but slowest for negative ones. HashSet

excels at negative queries, slightly outperforming even SSHash. CBL consistently maintains high throughput and low memory usage across both query types.



(a) Present k-mers.

(b) Absent k-mers.

Figure 11.7.: Time/memory trade-off when performing streaming queries of present/absent k-mers, for $k = 31$ and $p = 28$ bits; each point is averaged over a series of experiments.

11.4.3. Insertion and deletion

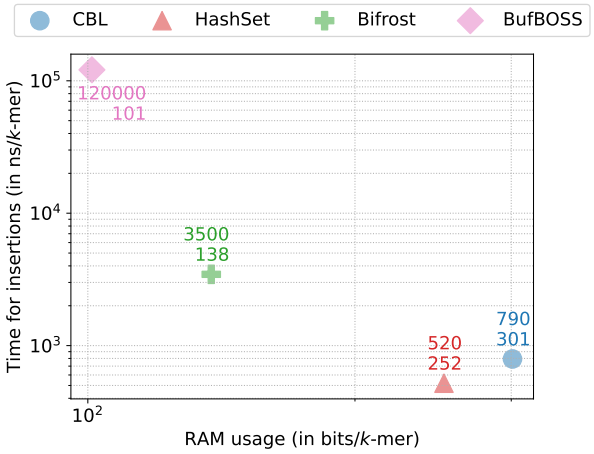
To assess update performance, we measure the cost of inserting and deleting k-mers; results are shown in Figure 11.8. Among the four tools supporting insertion, CBL and HashSet are the fastest, processing each k-mer in under a microsecond. Bifrost is approximately an order of magnitude slower, and BufBOSS another order of magnitude behind that. Memory usage is comparable across tools; BufBOSS is the most efficient, followed by Bifrost, HashSet, and CBL.

For deletion (not supported by Bifrost), the remaining tools show a similar pattern: CBL and HashSet lead in speed, BufBOSS lags by an order of magnitude, and memory usage is lowest for BufBOSS, then CBL, then HashSet.

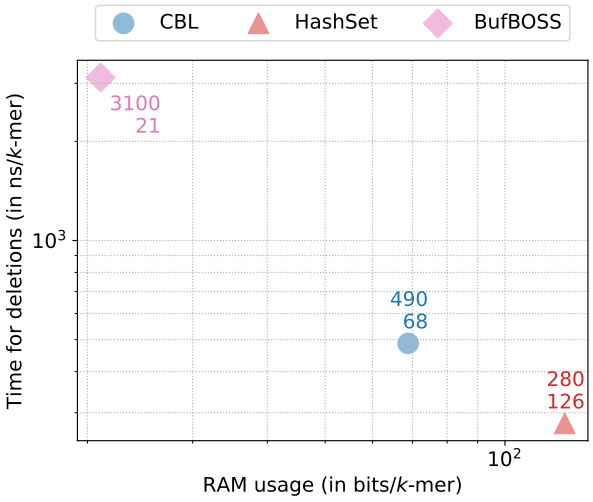
11.5. Perspectives

The use of necklaces as a k-mer representation is relatively recent, and many directions remain open. One open question concerns the rank of a necklace: the best known algorithm runs in $\mathcal{O}(k^2)$ time [SW17], which makes it impractical for large-scale use. A promising avenue would be to compute an approximate rank, mapping necklaces to a space slightly larger than $N(k)$ but still $\mathcal{O}(k)$ times smaller than the original k-mer space. Piecewise linear approximation, as used in the PLA-index [AM24], could provide such an approximation with controllable error, and would be a natural fit given the distribution of necklaces.

A second open question is the integration of super-necklaces into an actual index. We introduced the super-necklace representation and showed that it compactly encodes runs of consecutive necklaces sharing the same prefix, but this structure has not yet been connected to a data structure supporting efficient retrieval. Adapting the trie component of CBL to operate directly on super-necklaces is one concrete direction: within a run, successive suffixes differ by a single letter at a known position, which could allow the trie to share internal nodes more aggressively than it currently does.



(a) Insertions.



(b) Deletions.

Figure 11.8.: Time/memory trade-off when performing k-mer insertions/deletions (bottom), for $k = 31$ and $p = 28$ bits; each point is averaged over a series of experiments.

Regarding CBL itself, several implementation-level improvements are within reach. The current implementation is single-threaded, though suffix buckets are independent given a fixed prefix and could straightforwardly be distributed across threads. The core necklace computation and bucket scan operations are also good candidates for SIMD vectorization, and the techniques developed in Chapter 8 could serve as inspiration. The trie structure used for large buckets can also consume significant memory, which could be addressed by using a more compact alternative such as an adaptive radix tree [LKN13]. CBL being a fully dynamic structure also opens up possibilities beyond simple membership queries, which we explore in the next chapter.

12. Set operations on *k*-mer collections

> **Note**
>
> This chapter is partially adapted from Martayan et al. (2024), accepted to ISMB 2024 and published in Bioinformatics.
> The original paper reported benchmarks on set operations but did not describe the underlying algorithms. We present them here with refinements made since publication, together with new discussions on exponential search and connections to database query engines.

The previous chapter introduced CBL as a compressed, dynamic structure for exact k-mer set representation and established its competitive performance for membership queries, insertions, and deletions. We now turn to a class of operations that most k-mer structures do not support at all: set operations. Union, intersection, and difference are fundamental to pangenomic workflows: merging sequencing runs, identifying k-mers shared across all samples in a cohort, removing contaminant sequences. Yet they are typically performed either approximately, using lossy Bloom-filter-based methods [SBK22; LMCP22], or by reloading sets into a general-purpose hash table. At the time of writing, CBL was, to our knowledge, the only exact k-mer structure to offer in-place set operations. We explain what enabled this, what algorithmic machinery underlies it, and when the approach wins (or loses) against hash-based alternatives.

12.1. The sorted-iterator interface

The key enabler of efficient set operations in CBL is that its internal layout naturally yields a *sorted stream* of k-mers. Traversing the prefix bitvector in rank order and, for each occupied prefix, scanning its suffix bucket in turn produces all stored k-mers in lexicographic order. Each bucket stores suffixes as either a flat vector or, once it exceeds a size threshold, a trie. Tries are inherently sorted; only flat-vector buckets require an explicit sort, performed lazily when a set operation is requested. Since flat-vector buckets are small by definition, this per-bucket sort is cheap.

This sorted-iteration property transforms set operations into classical merge problems. Given two sorted streams, union and intersection can be computed in a single linear pass, consuming each stream once from left to right. By contrast, the hash-based alternative checks each element of one set against the other, requiring one random memory access per element. We examine the performance consequences of this difference in Section 12.4.

12.2. Multi-way set operations

CBL's set operations are implemented on top of `iter-set-ops` (available at https://github.com/imartayan/iter-set-ops), a Rust crate that performs intersection, union, and difference over an arbitrary number of sorted deduplicated iterators. Generalizing to $s > 2$ sets costs $\mathcal{O}(n \log s)$ rather than the $\mathcal{O}(ns)$ of repeated hash lookups, with no change to the algorithm structure. All operations are lazy: they return iterators that advance their inputs on demand, so the result can be streamed without materializing it in memory.

12.2.1. Union

The union of s sorted iterators is computed with a *binary heap-based s-way merge*. Each iterator contributes its current front element to a min-heap; at each step the minimum is extracted and its source iterator is advanced; if the new front matches the extracted value, that iterator is also advanced to suppress duplicates before inserting back into the heap. Total cost is $\mathcal{O}(n \log s)$ for s iterators of combined size n.

12.2.2. Intersection

Intersection uses a *max-index tracking* strategy rather than a heap. A "front" array holds the current element of each iterator. The algorithm records the index of the iterator holding the current maximum; all other iterators are advanced until they either match or overshoot that maximum. When all fronts are equal, the element is yielded and all iterators advance together.

This approach benefits from early termination: as soon as any iterator is exhausted, the computation stops. When the intersection is sparse, with few k-mers shared across all s sets, this happens well before the inputs are consumed and the actual cost falls well below the $\mathcal{O}(n \log s)$ worst case.

12.2.3. (Symmetric) difference

Set difference advances the left iterator and, for each left element, advances all right iterators until their fronts are $\geq$ the left element; if any right front equals the left element it is suppressed. This runs in $\mathcal{O}(n + m)$ for a left set of size n and right sets of combined size m.

12.3. Prefix-level pruning

CBL exploits its quotiented layout to perform set operations at the *prefix level* before descending to suffixes. When scanning two CBL structures in tandem, both prefix bitvectors are advanced simultaneously. If one structure has no bucket for a given prefix rank, the entire prefix range is skipped using a single rank/select query, with no suffix work performed. Only when both structures share an occupied prefix do their suffix buckets enter the merge or intersection routine.

Because necklace values are skewed towards small values (as discussed in Chapter 11), many prefix slots are empty, and this pruning eliminates a large fraction of the work in practice. When k-mer sets are biologically related, their occupied prefix ranges overlap heavily, so the relative saving shifts to the suffix level. Identical necklaces are then handled by a zero-cost path that advances both iterators simultaneously. In-place mode further reduces peak memory by modifying one input structure directly, avoiding the allocation of a third.

12.4. Benchmarks

To evaluate the practical impact of sorted iteration, we compare CBL against HashSet, a hash table based on Google's Swiss Tables, on intersection and union of k-mer sets built from an expanding collection of bacterial genomes, ranging from 10^7 to 10^9 k-mers ($k = 31$, $p = 28$ bits). Time and memory results for intersection are shown in Figure 12.1, and for union in Figure 12.2.

CBL achieves 200 ns per k-mer on intersection and 500 ns per k-mer on union; HashSet requires 400 ns and 900 ns respectively. The memory advantage is larger: CBL's in-place operations do not allocate a result structure, while HashSet must construct a third hash table. Comparable trends hold for set difference and symmetric difference. These results reflect a broader pattern: at the scale typical in genomics, with large sets streamed across many samples, sorted iteration consistently outperforms hash-based alternatives.

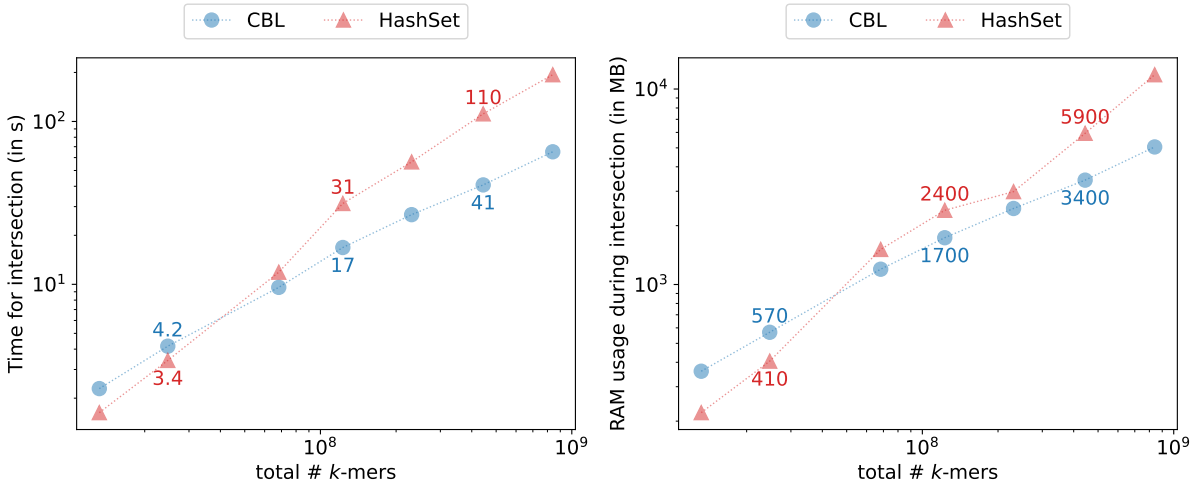


Figure 12.1.: Time and memory trade-off when performing set intersections, for $k = 31$ and $p = 28$ bits.

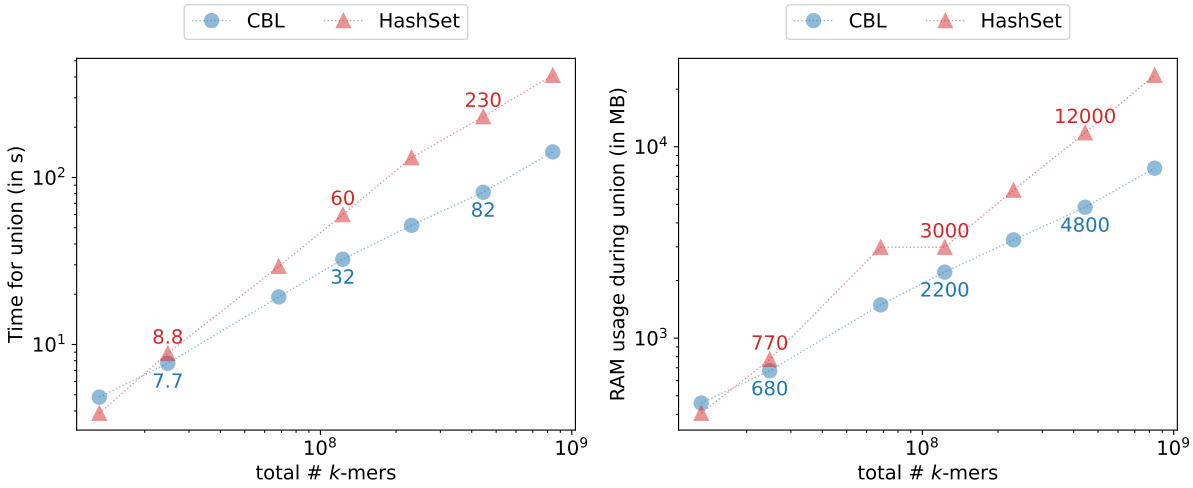


Figure 12.2.: Time and memory trade-off when performing set union, for $k = 31$ and $p = 28$ bits.

12.5. Exponential search for skewed set sizes

The linear scan of merge-based algorithms is efficient when the two sets are of comparable size, but wasteful when one set is much smaller. Consider intersecting a small query set of m k-mers against a large index of n k-mers with $m \ll n$: iterating through all n elements to find m matches is suboptimal.

Exponential search [BY76] addresses this: rather than advancing one step at a time in the large iterator, double the stride until overshooting the target, then binary-search within the overshot interval. The cost per match found is $\mathcal{O}(\log d)$ where d is the gap to the next match. Demaine et al. [DLM00] extended this to multi-way intersection. For two sets of sizes $m \ll n$, the cost specializes to $\mathcal{O}(m \log(n/m))$, a significant improvement over the $\mathcal{O}(n+m)$ linear scan. An alternative achieving the same average complexity is the double binary search of Baeza-Yates and Salinger [BS10]: binary-search the median of the smaller set in the larger one, then recurse on both halves. In practice both approaches outperform linear merge when the size ratio is large.

The `iter-set-ops` library uses max-index tracking for intersection, advancing all iterators towards the current maximum, but without the exponential stride; it therefore remains $\mathcal{O}(n+m)$ in the two-set case. Adding exponential search would reduce this to $\mathcal{O}(m \log(n/m))$ for skewed sizes. The gain is largest whenever set sizes are strongly skewed: a small query intersected against a large reference, a small contaminant set removed from a large assembly, or a multi-way intersection across very unequally-sized sets. In the last case, the largest iterator should always be the one advanced with exponential jumps. Applying this to CBL would however require random access into suffix buckets to binary-search within the overshot interval, which is incompatible with the trie representation currently used for large buckets. Supporting exponential search would therefore require redesigning large buckets with a structure that supports both sorted iteration and random access.

12.6. Perspectives from database query engines

The algorithmic landscape of k-mer set operations has strong connections to classical database join strategies [Gra93], suggesting that techniques developed for query engines could benefit k-mer set processing. CBL's sorted-iterator approach resembles *sort-merge join*, while HashSet resembles *hash join*. When one set is very small, the pattern corresponds to *index-nested loop join*: iterate over the small set and perform membership queries on the large index, using exponential search to skip large gaps. CBL's prefix-level pruning is analogous to *partition pruning* in query optimizers.

A related opportunity comes from the information-retrieval literature: Lemire et al. [LBK15] showed that SIMD-accelerated intersection nearly doubles throughput on inverted-index posting lists, a setting structurally similar to k-mer set intersection on sorted lists. More broadly, adaptive strategy selection and partition-parallel processing have precedent in the database literature but remain unexplored for k-mer sets. The structural similarities between k-mer set operations and relational query processing suggest that this literature is a productive source of inspiration for future work.

13. Dynamic super-*k*-mer maps

> **Note**
>
> This chapter is partially adapted from Smith et al. (2024), currently under review. Yoann Dufresne and Antoine Limasset designed the original prototype. Caleb Smith improved and completed the implementation. We discussed algorithmic improvements together and I wrote experiments scripts. The final section presents my ongoing work on deduplication strategies, not part of the original paper.

Section 4.3 introduced super-*k*-mers as a natural byproduct of minimizer-based parsing: consecutive k-mers sharing a minimizer are grouped into a single string, reducing both the number of objects to index and the memory needed to represent each one. This chapter asks whether super-*k*-mers can serve as the primary representation inside a *dynamic* k-mer dictionary, replacing individual k-mer entries altogether.

Using super-*k*-mers as a full dictionary representation, however, raises two practical challenges. The first is lookup: given a k-mer, finding it efficiently inside a bucket of super-*k*-mers is not straightforward, because sorting super-*k*-mers lexicographically does not help locate a specific k-mer. The second is load balancing: minimizer partitions are highly non-uniform on real genomic data, so a handful of very large buckets can dominate overall query time. The main contributions of this chapter are a super-*k*-mer encoding called the *interleaved transform* that enables binary search within a bucket, and a superbucket scheme that flattens the skewed minimizer distribution. We also place the super-*k*-mer-level approach in relation to the k-mer-level approach of the previous chapter. The interleaved transform and superbucket scheme are implemented in Brisk, available at https://github.com/Malfoy/Brisk.

13.1. Interleaved super-*k*-mers

13.1.1. Lazy encoding of minimizers

The dictionary is organized as a collection of buckets, one per minimizer: each k-mer is routed to the bucket of its minimizer and stored implicitly inside one of the super-*k*-mers held there. Since all super-*k*-mers in a bucket share the same minimizer, we can avoid storing that minimizer in each super-*k*-mer. In a maximal super-*k*-mer, the position of the minimizer within the sequence is determined by the super-*k*-mer's length, so the minimizer can be omitted from the encoding and reconstructed on demand. This *lazy encoding* lowers the per-*k*-mer bit cost: a maximal super-*k*-mer of length $2k - m$ bases encodes on average $(w + 1)/2$ k-mers, giving a ratio of $\frac{8(k-m)}{w+1} = 8\left(1 - \frac{1}{w+1}\right)$ bits per k-mer, which for practical values of k and m is slightly below 8 bits.

To simplify memory management and allow constant-time access to any super-*k*-mer in a list, every super-*k*-mer is allocated at its maximum possible length of $2k - m$ bases, with bases packed at 2 bits each. The actual extent of each super-*k*-mer is recorded in two small fields, its number of k-mers and the position of its minimizer within it. The first determines the super-*k*-mer's total length, and together with the second they fix the number of bases on each side of the minimizer, so positions beyond either side are known and can be treated as a sentinel N when needed. This wastes some space compared to a tightly packed variable-length encoding, but avoids storing

per-super-k-mer offsets and makes the list amenable to binary search.

13.1.2. The bucket lookup problem

Looking up a k-mer in its bucket means checking whether it appears in any of the super-k-mers stored there. A k-mer has at most one possible position within a given super-k-mer, determined by the relative positions of their minimizers, so each individual comparison is cheap. The difficulty is that buckets can be very large, so a linear scan may end up being costly.

To use binary search, we need to sort the super-k-mers. However, standard lexicographic order does not help here: it weights the outermost bases first, which are the least shared among the k-mers of a super-k-mer. The leftmost base of a maximal super-k-mer belongs to only one of its k-mers, the next to two, and so on inward toward the minimizer. Effective binary search needs the discriminating bases to come first, but lexicographic order puts them last.

13.1.3. Interleaving

We propose a base reordering, the *interleaved transform*, that addresses this. The interleaved form of a super-k-mer starts with the minimizer sequence, then alternates between the base immediately to its left and the base immediately to its right, expanding outward until all positions are exhausted. Positions with no base (because the super-k-mer does not extend as far in one direction) are filled with the character N. Figure 13.1 shows examples of this transform.



Figure 13.1.: Toy example of the AAAAAAA bucket associated to four super-k-mers in their interleaved representation.

The same transform applies to a k-mer, viewed as a super-k-mer of minimal length, and gives a key property: a k-mer is contained in a super-k-mer if and only if its interleaved form is a prefix of the interleaved super-k-mer, with N matching any character. To see why, note that interleaving lists positions outward from the minimizer. A k-mer contained in a super-k-mer extends in both directions no further than the super-k-mer does, so its positions come first in the interleaved order. Its bases therefore form a prefix of the super-k-mer's interleaved form, with N filling the positions beyond the k-mer's own extent.

Given this property, super-k-mers can be sorted lexicographically in their interleaved form and queried by binary search. Each non-N base in the query k-mer narrows the search space by a factor of up to four, so in most cases the bucket is searched in $\mathcal{O}(\log S)$ steps rather than $\mathcal{O}(S)$. Figure 13.2 illustrates several example searches in a bucket of 32 super-k-mers.

13.1.4. Limitations

Because N must match any character, a step involving an N does not reduce the search space, and all four sub-searches must be pursued in parallel from that point. This is harmless when N characters appear only at the end of the query, after the search space has already been reduced to a small number of candidates. It is costly when the minimizer sits at the very start or end of the k-mer, which places N characters at the beginning of the interleaved form and prevents any early reduction of the search space, as in k-mer 2 and 3 of Figure 13.2. Such k-mers degrade to near-linear probing. In practice, the minimizer occupies an interior position for the large

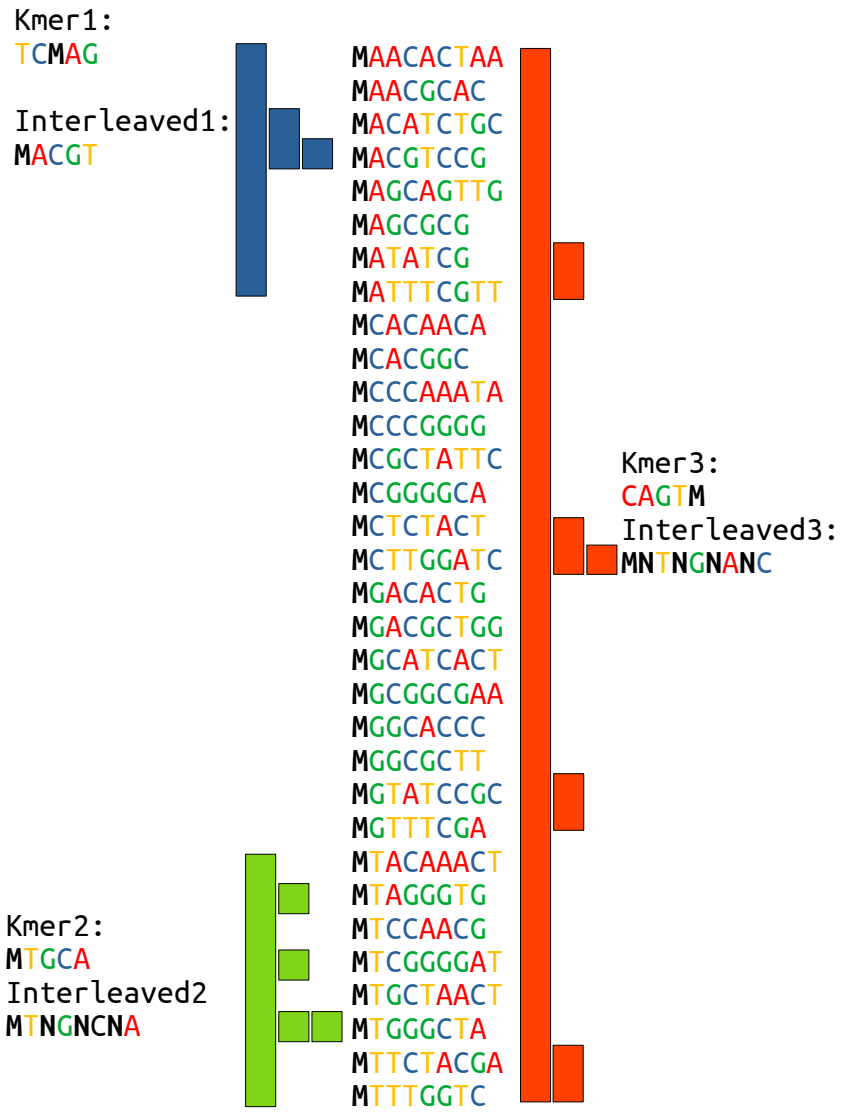


Figure 13.2.: Examples of k-mer search in a bucket of 32 super-k-mers. M denotes the bucket minimizer. Colored bars show the successive search spaces after each step. k-mer 1 (blue) reaches a unique match quickly. k-mer 2 (green) encounters an N mid-search and briefly stalls before the next base resolves the search. k-mer 3 (red) starts with an N and must explore all four branches at the first step, recovering only after a discriminating base is found.

majority of k-mers, so this worst case is rare, arising mainly at sequence boundaries or with very small window sizes.

13.2. Load balancing and superbuckets

13.2.1. Skewed bucket sizes

The interleaved sort provides sublinear lookup within a bucket, but the benefit is diluted when bucket sizes are highly skewed. As established in Chapter 10, minimizer partition sizes are highly non-uniform in practice: a small fraction of minimizers attract a disproportionately large number of super-k-mers, and these very large buckets can dominate overall query time even when the average size is moderate. The challenge is to flatten this distribution without breaking the bucket abstraction that makes binary search possible.

13.2.2. Superbuckets via bijective hashing

Applying a hash function to minimizer values would spread them uniformly across buckets, but a surjective hash introduces collisions between distinct minimizers, which is incompatible with an exact structure. We use a bijective hash instead, a reversible permutation over minimizer values, which achieves the same uniformity while letting us recover the original minimizer from its hashed value.

We then group the 4^m possible minimizer buckets into 4^b superbuckets by using the first b bits of each hashed minimizer value as the superbucket index. Each superbucket aggregates 4^{m-b} original buckets, and because the permutation distributes minimizer values uniformly, superbucket sizes are far less skewed than individual bucket sizes. Figure 13.3 shows the successive construction steps.

Within a superbucket, super-k-mers from different minimizer buckets remain distinguishable because the full interleaved form starts with the hashed minimizer, which is distinct across buckets. Binary search therefore traverses the minimizer prefix at no extra cost before entering the bucket-specific portion of the sorted list.



Figure 13.3.: Construction steps of a superbucket. Minimizers are highlighted in yellow. Plain super-k-mers (1) are rewritten in interleaved form (2), the minimizer is replaced by its hashed value (3), the common superbucket prefix is made implicit (4), and the list is sorted (5).

13.2.3. Sorting and buffering

Maintaining a strictly sorted list on every insertion would require an $\mathcal{O}(S)$ shift per operation, which becomes expensive as buckets grow. A practical solution is to keep a small unsorted buffer alongside the sorted list. New super-k-mers enter the buffer and are scanned linearly until 1000 new entries have accumulated, at which point only the new chunk is sorted and merged into the existing sorted portion via `std::inplace_merge`. This amortizes sorting cost across many insertions without changing asymptotic lookup time, and avoids ever re-sorting the full list. A

larger threshold reduces sorting frequency and speeds up insertion, but slows queries because more candidates must be scanned linearly before the sorted portion is searched. The threshold therefore acts as a tradeoff between insertion and query throughput.

13.3. Super-*k*-mer versus *k*-mer level representation

The structure presented in Chapter 11 for CBL and the one described in this chapter represent two distinct approaches to organizing a dynamic k-mer dictionary: one stores each k-mer as an independent record, the other groups them implicitly inside super-k-mers.

In a k-mer-level structure, each k-mer is an independent record, and insertion, lookup, and deletion each act on exactly one entry. This conceptual simplicity also enables straightforward optimizations: as discussed in Chapter 12, CBL's sorted-iterator layout supports efficient in-place set operations while remaining memory-efficient. Because every k-mer maps to a unique canonical representative, deduplication is structural rather than a separate concern.

In a super-k-mer-level structure, k-mers are stored implicitly inside longer strings. The benefit is that neighboring k-mers sharing a minimizer are encoded together, and the shared minimizer is not repeated. This gives a lower bits-per-k-mer footprint, and insertion is streaming-friendly since a new super-k-mer is placed directly into the appropriate bucket without decomposing it into individual k-mers.

The main cost is that the same k-mer can in principle appear in multiple super-k-mers, typically when the same region is covered by overlapping reads. In CBL, deduplication is inherent since each k-mer maps to a unique necklace and a unique bucket entry. In a super-k-mer structure, deduplication requires either a membership test before every insertion (which partially erases the throughput advantage) or a post-processing step that is incompatible with streaming use. This remains an open problem and the primary structural limitation of the super-k-mer-level approach. We explore several directions for addressing it in Section 13.5.

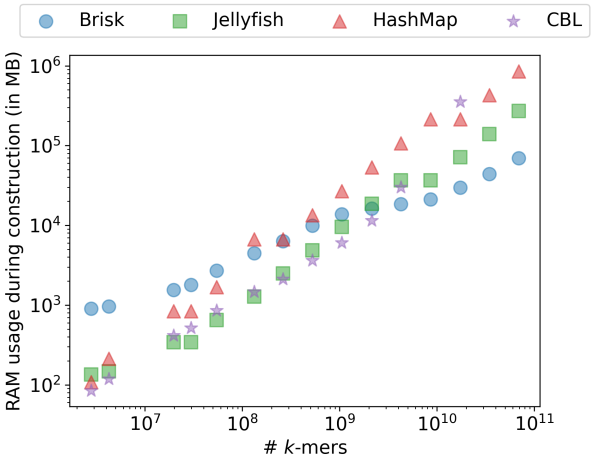
A second consideration is the granularity of per-k-mer operations. In a k-mer-level structure, the annotation or count associated with a single k-mer can be updated in isolation. In a super-k-mer-level structure, accessing that annotation requires locating the k-mer within its containing super-k-mer, which is a lookup rather than a direct address. For applications where every k-mer is visited at most once (graph traversal, read mapping), this difference is negligible. For applications that revisit k-mers frequently or require deletion, the k-mer-level approach is more natural.

13.4. Performance in practice

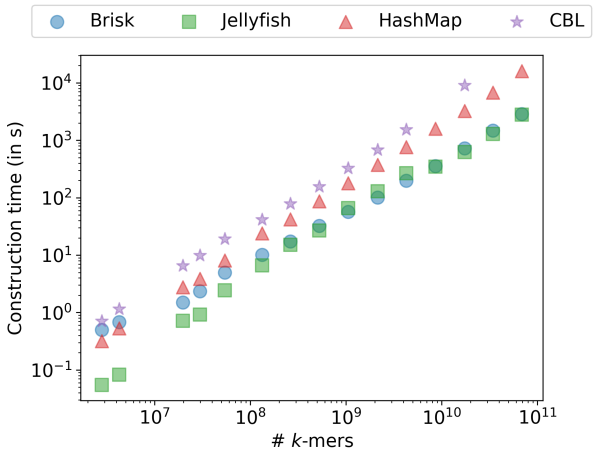
We compare Brisk against three reference structures: Jellyfish [MK11], a popular k-mer counter built on an efficient hash table; CBL [MCLM24], the dynamic k-mer set structure discussed in Chapter 11; and the standard Rust `HashMap`, used as a baseline for general-purpose dynamic dictionaries. Benchmarks were performed on growing collections of bacterial genomes (2^0 to 2^{14} genomes), at two k-mer sizes: $k = 31$, a common choice that fits in a 64-bit integer, and $k = 59$, the maximum size supported by CBL.

Figure 13.4 shows the memory footprint and construction time across all four tools. For both k-mer sizes, Brisk uses substantially less memory than the reference structures, and the gap widens at $k = 59$ as hash-based methods store proportionally more bits per k-mer while Brisk's lazy encoding amortizes the cost across each super-k-mer. At $k = 31$, Brisk matches Jellyfish's throughput. At $k = 59$ it becomes the fastest of the four.

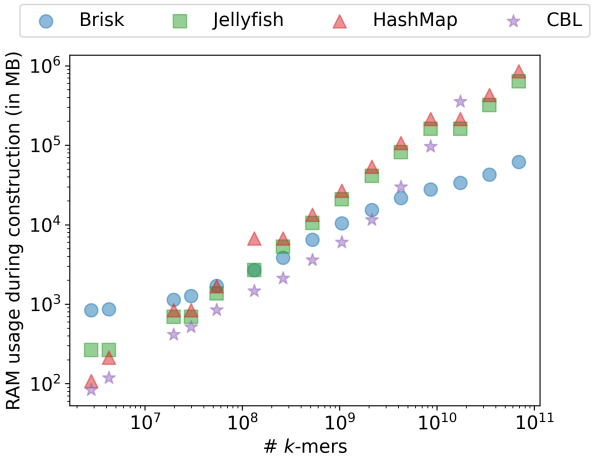
The multi-threaded comparison is not entirely fair, however: Brisk and Jellyfish parallelize, whereas CBL and the Rust `HashMap` do not. Figure 13.5 isolates this by repeating the benchmark on a single core. The picture is qualitatively similar: Brisk is slightly slower than the Rust `HashMap` at $k = 31$ but remains the fastest indexer at $k = 59$.



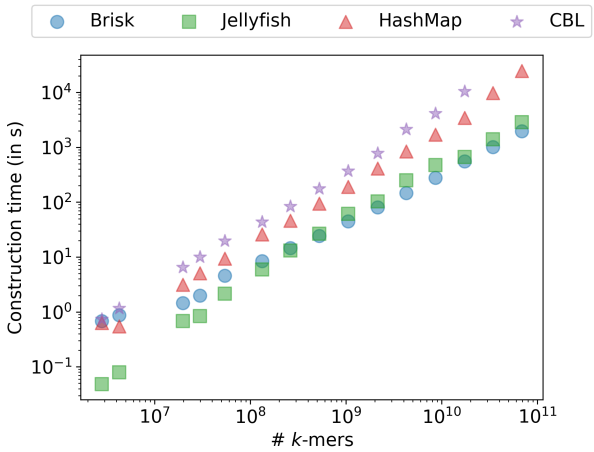
(a) Memory usage for $k = 31$



(b) Construction time for $k = 31$

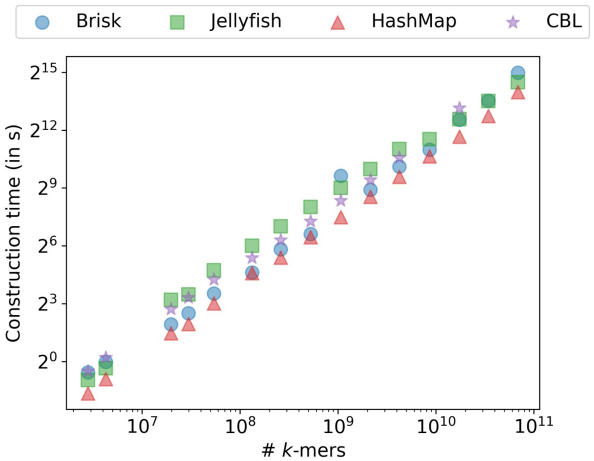


(c) Memory usage for $k = 59$

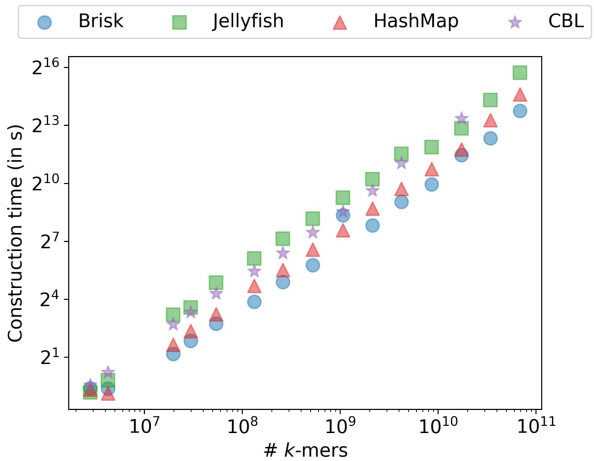


(d) Construction time for $k = 59$

Figure 13.4.: Multi-threaded performance of Brisk, Jellyfish, CBL, and Rust `HashMap` across growing collections of bacterial genomes, for $k = 31$ (top) and $k = 59$ (bottom).



(a) Single-threaded construction time for $k = 31$



(b) Single-threaded construction time for $k = 59$

Figure 13.5.: Single-threaded construction time of Brisk, Jellyfish, CBL, and Rust `HashMap`, controlling for the fact that CBL and Rust `HashMap` are not parallelized.

13.5. Toward streaming deduplication

The structural limitation of super-k-mer dictionaries identified above stems from a single underlying cause: testing which k-mers of a query are already present requires locating them one by one inside the containing super-k-mers. A natural way to bypass this is to express every such operation as a check on the prefixes and suffixes of super-k-mers themselves, without materializing individual k-mers. I am exploring this direction in an ongoing work, prototyping a dynamic super-k-mer dictionary that supports on-the-fly deduplication and compaction.

13.5.1. A compact super-k-mer representation

The approach I propose packs each super-k-mer into a compact two-word encoding. The first word holds the minimizer along with the prefix and suffix lengths. The second word holds the prefix and suffix bases interleaved at the bit-pair level: positions in the word alternate between left bases and right bases, with 2 bits per base. This fits up to 16 bases on each side of the minimizer, giving the constraint $k - m \leq 16$.

In this layout, comparing two super-k-mers that share the same minimizer reduces to a single XOR between their encodings, followed by independent leading-zero counts over the left and right halves. The result is a pair (p_L, p_R), the lengths of the common interleaved prefix on each side, computed in a few cycles. The same representation supports constant-time truncation of either side and stitching the left side of one super-k-mer with the right side of another. These operations are enough to manipulate super-k-mers without ever scanning bases individually.

13.5.2. Coverage-based insertion

Given a query super-k-mer Q, the goal at insertion time is to determine which of its k-mers are already present in the bucket, and only store the new ones. The candidates to compare against are the bucket's super-k-mers that share Q's minimizer, found by binary search on the sorted prefix of the bucket. For each candidate S, the comparison above returns a pair (p_L, p_R) measuring how far the two super-k-mers agree on each side of the minimizer. If $p_L + p_R \geq k - m$, the agreement spans at least one full k-mer, so S contains a contiguous range of Q's k-mers, determined by where the match starts and ends relative to the minimizer.

The covered ranges from all candidates are combined into a single coverage of Q's k-mers. If every k-mer is covered, the insertion is skipped entirely. Otherwise, for each uncovered range, the insertion either extends an existing super-k-mer (using the side-stitching operations above) to incorporate the missing k-mers, or stores a truncated copy of Q spanning only that range.

Since the bucket is already scanned to find the matching minimizer, computing the coverage adds only a few bit operations per candidate. Insertion-time deduplication therefore preserves the streaming property of the original super-k-mer-level approach, with no separate post-processing pass.

13.5.3. Open extensions

My ongoing prototype implements insertion-time deduplication, which already addresses the main structural limitation discussed above. Several extensions remain open work. Deletion has no streaming equivalent of the coverage mechanism, since removing k-mers from the middle of a super-k-mer requires splitting it. Set operations beyond insertion, in particular union and intersection over two indexes, can be expressed in principle as bucket-by-bucket coverage computations, but the interleaved representation does not yet expose the necessary primitives to enumerate shared k-mers across two sorted super-k-mer lists efficiently. Finally, dynamic resizing of the superbucket layout to absorb growing datasets without rebuilding the index is also under investigation.

14. From super-k-mers to hyper-k-mers

> **Note**
>
> This chapter is adapted from Martayan et al. (2025), accepted to RECOMB 2025 and to be published.
> Lucas Robidou and I contributed equally to this work.

The previous chapters built k-mer dictionaries on top of contiguous super-k-mers, one bucket per minimizer. At the k-mer sizes most prior work is calibrated for, around $k = 31$, the per-k-mer cost of this layout is moderate. It degrades as k grows: each super-k-mer carries $k - 1$ bases of overlap with each of its two neighbors, so a super-k-mer representation asymptotes to 6 bits per nucleotide as discussed in Section 4.3, three times the 2-bit lower bound. Static representations such as unitigs and eulertigs do reach 2 bits per nucleotide, but they are built once from a finalized k-mer set, require a separate counting pass, and dissolve the per-minimizer partitioning that the structures of Chapter 13 rely on for cache-friendly indexing and load balancing.

This regime of large k-mer sizes is becoming practically relevant, as long-read technologies now reach error rates below 1% for Oxford Nanopore [Ser+22] and 0.1% for PacBio HiFi [Ban+22], which makes k-mer sizes in the hundreds tractable and useful for genome assembly, pangenomic comparisons, and fine-grained k-mer analyses.

In this chapter, we introduce a representation called *hyper-k-mers* that aims to keep the streaming, partitioned nature of super-k-mers while approaching the asymptotic efficiency of static representations. The core idea, developed formally in Section 14.2, is to share each $k - 1$ base overlap between consecutive super-k-mers instead of duplicating it on each side, bringing the cost from 6 down to 4 bits per nucleotide at large k.

We put hyper-k-mers to work in KFC, a streaming k-mer counter available at https://github.com/lrobidou/KFC that uses hyper-k-mers as primary storage. We describe the counting algorithm in Section 14.3 and compare KFC against existing k-mer counters in Section 14.4.

14.1. Space analysis of super-k-mers and closed syncmers

We use the notation of Chapter 4 throughout: S is a random genomic string over $\Sigma = \{A, C, G, T\}$ with k-mer size k, minimizer size $m < k$, window size $w = k - m + 1$, and minimizer scheme density d, with random minimizers satisfying $d = 2/(w+1)$. We add one new notion, the overlap between consecutive super-k-mers.

Definition 14.1 (Overlap of consecutive super-k-mers)**.** Given two consecutive super-k-mers u and v, we define the overlap of u and v, denoted $\mathrm{ov}(u, v)$, as the $k - 1$ bases shared by the rightmost k-mer of u and the leftmost k-mer of v.

This section establishes the asymptotic space cost of super-k-mers and closed syncmers, the two reference points that the hyper-k-mer analysis in Section 14.2 compares against. Super-k-mers are the direct baseline that hyper-k-mers improve on. Closed syncmers, while not a stand-alone k-mer-set representation, are included as a comparison point: they pave sequences better than minimizer-based sampling [Edg21; SBK22] and reach 4 bits per nucleotide.

14.1.1. Super-*k*-mers

Lemma 14.1 (Average number of super-k-mers)**.** Given a random string S, the expected number of super-k-mers is $(|S| - m + 1) \times d$.

Proof. By definition of the density d, the expected number of minimizers is $(|S| - m + 1) \times d$. Since each minimizer corresponds to one super-k-mer, the expected number of super-k-mers is the same. □

Lemma 14.2 (Average length of super-k-mers)**.** Given a random string S, the average length of a super-k-mer is $\frac{|S|-k+1}{(|S|-m+1)\times d} + k - 1$. In particular, it approaches $\frac{1}{d} + k - 1$ as $|S| \to +\infty$.

Proof. By Lemma 14.1, there are $(|S|-m+1)\times d$ super-k-mers on average. Since there are $|S|-k+1$ k-mers in S, each super-k-mer contains $\frac{|S|-k+1}{(|S|-m+1)\times d}$ k-mers on average, i.e. $\frac{|S|-k+1}{(|S|-m+1)\times d}+k-1$ bases. □

Theorem 14.1 (Super-k-mer space usage)**.** Given a random string S, the expected total number of bases in the super-k-mers of S is asymptotically equivalent to $|S|\cdot(1+(k-1)\cdot d)$ as $|S| \to +\infty$.

Proof. Let S_{sk} denote the set of super-k-mers of S, $|S_{sk}|$ the number of super-k-mers in S and $S_{sk}[i]$ denote the i-th super-k-mer of S. The total number of characters in S_{sk} is:

$$
\begin{aligned}
\sum_{i=0}^{i<|S_{sk}|} |S_{sk}[i]| &= |S_{sk}| \times \sum_{i=0}^{i<|S_{sk}|} \frac{|S_{sk}[i]|}{|S_{sk}|} \\
&= (|S| - m + 1) \times d \times \sum_{i=0}^{i<|S_{sk}|} \frac{|S_{sk}[i]|}{|S_{sk}|} \\
&= (|S| - m + 1) \times d \times \left(\frac{|S| - k + 1}{(|S| - m + 1) \times d} + k - 1 \right) \\
&\underset{|S|\to+\infty}{\sim} |S| \times d \times \left(\frac{1}{d} + k - 1 \right) \\
&= |S| \cdot (1 + (k-1) \cdot d)
\end{aligned}
$$

□

In particular, for random minimizers which have a density $d = 2/(w+1) = 2/(k-m+2)$, the average number of bases in a set of super-k-mers is asymptotically equivalent to $|S| \cdot \left(1 + \frac{2(k-1)}{k-m+2}\right) \sim 3 \cdot |S|$ when $k \gg m$. Thus, a super-k-mer scheme requires 3 bases per nucleotide in $|S|$, i.e. 6 bits per nucleotide in $|S|$ for an alphabet of size 4.

14.1.2. Closed syncmers

Recall from Section 4.4 that a closed syncmer is a k-mer whose minimizer sits at its first or last position.

Theorem 14.2 (Closed syncmer space usage)**.** Given a random string S, the expected total number of bases in the closed syncmers of S is asymptotically equivalent to $|S|\cdot k\cdot d$ as $|S| \to +\infty$.

Proof. Each k-mer consists of $w = k - m + 1$ m-mers. Let us consider the last $w + 1$ m-mers after spawning a new minimizer. Since a new minimizer has just been found, the smallest m-mer, among the last $w + 1$, is either the first (if the minimizer changed because the previous one went out of scope) or the last (if the new minimizer is smaller than the previous one). If the smallest m-mer is the first one, then the previous k-mer is a closed syncmer. Otherwise, the smallest m-mer is at the end of the last k-mer, which is thus a closed syncmer. Thus, except the first minimizer, each minimizer creates a closed syncmer, so there are $(|S| - m + 1) \times d$

closed syncmers, on average, in S. Since each closed syncmer has length k, the expected total number of bases is $(|S| - m + 1) \times d \times k$, which is asymptotically equivalent to $|S| \cdot k \cdot d$ as $|S| \to +\infty$. □

In particular, for random minimizers, the number of bases in a set of closed syncmers is asymptotically equivalent to $|S| \cdot \frac{2 \cdot k}{k-m+2} \sim 2 \cdot |S|$, for $k \gg m$. As such, closed syncmers require 2 bases per nucleotide in $|S|$, i.e. 4 bits for an alphabet of size 4.

14.2. Hyper-*k*-mers

Definition 14.2 (Hyper-k-mer)**.** Given three consecutive super-k-mers u, v and w, let m_v be the minimizer of v. We define the hyper-k-mer associated to m_v as $(\mathrm{ov}(u, v), \mathrm{ov}(v, w))$, and refer to $\mathrm{ov}(u, v)$ as its left part and $\mathrm{ov}(v, w)$ as its right part. By convention, if v has no predecessor in the sequence, we define its left part as the first $k - 1$ bases and if it has no successor, we define its right part as the last $k - 1$ bases.

We can already make a few observations based on this definition: every hyper-k-mer contains exactly $2(k - 1)$ bases and given two consecutive hyper-k-mers x and y, the right part of x and the left part of y are identical. This means that we can effectively save $k - 1$ bases for each pair of consecutive hyper-k-mers. Figure 14.1 gives an example of hyper-k-mers and shows that their overlapping parts can be shared.



Figure 14.1.: Example of minimizers, super-k-mers and hyper-k-mers for $m = 3$ and $k = 11$ using lexicographic order. In this example, the original sequence contains 40 bases and super-k-mers use 70 bases while hyper-k-mers use 50 bases.

Theorem 14.3 (Hyper-k-mer space usage)**.** Given a random string S, the expected total number of bases in the hyper-k-mers of S is asymptotically equivalent to $|S| \cdot (k-1) \cdot d$ as $|S| \to +\infty$. In addition, storing the links between the left and right parts results in a $2 \log_2(d|S|)$ bits overhead for each hyper-k-mer.

Proof. Each minimizer of S is associated with one hyper-k-mer, so there are $(|S| - m + 1) \times d$ hyper-k-mers on average. What's more, each pair of consecutive hyper-k-mers shares $k - 1$ bases which can be written only once. Thus, except for the first hyper-k-mer (which uses $2(k - 1)$ bases), each hyper-k-mer uses $k - 1$ bases. Therefore, the expected total number of bases is $((|S|-m+1) \times d+1) \times (k-1)$, which is asymptotically equivalent to $|S| \cdot (k-1) \cdot d$ as $|S| \to +\infty$.

Finally, we need to maintain a "link" between the two parts of each hyper-k-mer by storing their indices. Since the expected number of hyper-k-mers is equivalent to $d|S|$, each index takes $\log_2(d|S|)$ bits of space, leading to an overhead of $2\log_2(d|S|)$ bits for each hyper-k-mer. □

In particular, for random minimizers, the number of bases in a set of hyper-k-mers is asymptotically equivalent to $|S| \cdot \frac{2\cdot(k-1)}{k-m+2} \sim 2$ bases per nucleotide in $|S|$, for $k \gg m$. Moreover, assuming $m = \Omega(\log_2(d|S|))$, the space overhead of the links becomes negligible as $k \gg m$. Figure 14.2 gives an overview of the evolution of the space usage as k increases, and compares it to the evolution of super-k-mers' space usage.

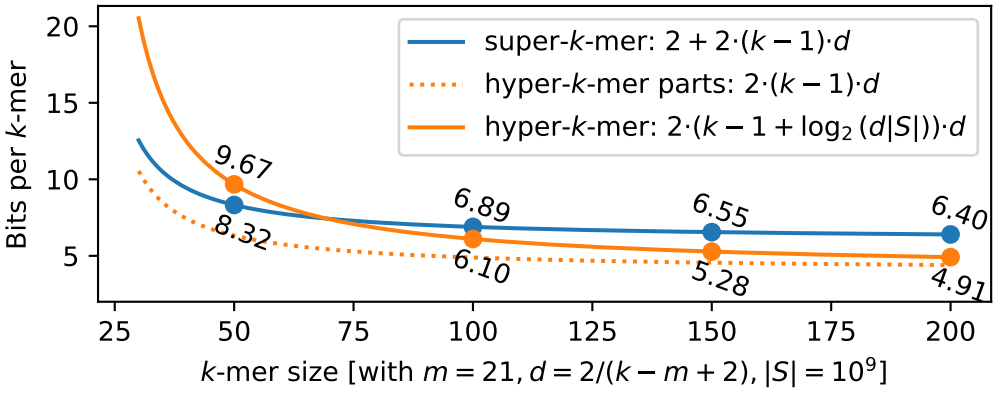


Figure 14.2.: Space usage of super-k-mers and hyper-k-mers for $k \in [30, 200]$, with random minimizers of size 21.

These results, summarized in Table 14.1, show that hyper-k-mers are more space-efficient than super-k-mers regardless of the chosen minimizer scheme, and reach the same 4 bits per nucleotide asymptote as closed syncmers up to the logarithmic link overhead, while remaining a stand-alone k-mer-set representation. In particular, hyper-k-mers require $\frac{2}{3}$ of the space used by super-k-mers for random minimizers, and this ratio gets even lower for minimizer schemes with a lower density.

Table 14.1.: Comparison of the space usage of super-k-mers, closed syncmers and hyper-k-mers.

Representation	Expected number of bits / nucleotide	For random minimizers with $k \gg m = \Omega(\log_2(d\|S\|))$
Super-k-mers (Theorem 14.1)	$2 + 2 \cdot (k-1) \cdot d$	6
Closed syncmers (Theorem 14.2)	$2 \cdot k \cdot d$	4
Hyper-k-mers (Theorem 14.3)	$2 \cdot (k-1) \cdot d + 2\log_2(d\|S\|) \cdot d$	4

14.3. Counting k-mers using hyper-k-mers

This section proposes k-mer counting as an application of hyper-k-mers to a real-life use case. Hyper-k-mers are used as a proxy for storing k-mer counts. A simplified overview of the resulting index is presented in Figure 14.3.

Our k-mer counter implements a two-pass algorithm. In the first stage, we iterate over super-k-mers in the input sequences. We identify which super-k-mers are solid, i.e. are present

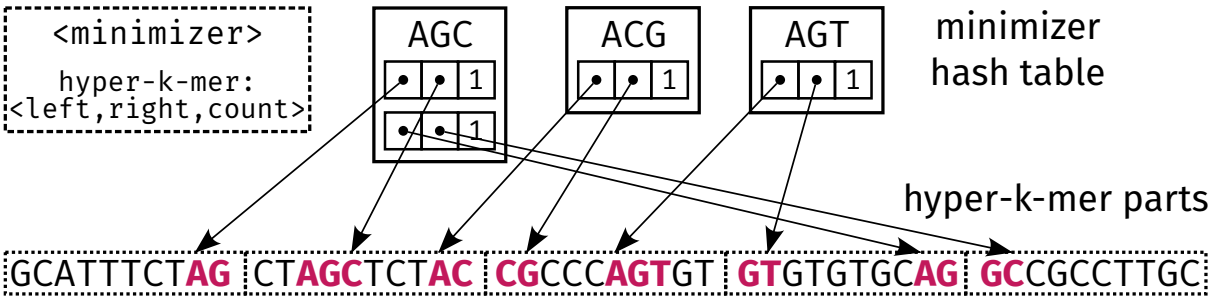


Figure 14.3.: High-level view of the KFC data structure, using the example introduced in Figure 14.1. Two data structures are maintained. **Bottom**: a vector of hyper-k-mer parts stores the left and right parts of the hyper-k-mers computed from the original set of sequences. The bold parts correspond to the (possibly truncated) minimizers of the sequence. **Top**: a hash table associates each minimizer to a list of entries. Each entry represents a hyper-k-mer and its count. Each hyper-k-mer is represented using pointers (represented as dots) to its left and right parts in the vector of hyper-k-mer parts. To retrieve the parts, each pointer consists of the part's position in the vector, a start and end position in that part, and an orientation flag to handle reverse-complements.

more than twice in the input. If a super-k-mer is solid, the hyper-k-mer, along with its minimizer, is computed using its left and right super-k-mer neighbors (see Definition 14.2). If a computed hyper-k-mer was already present in the index, we increase its count in the hash table, otherwise, we insert a new entry in the table and insert the corresponding hyper-k-mers in the vector (encoded using two bits per base). However, doing so prevents us from counting the first and last super-k-mers of a read, as they are likely truncated and, by extension, unique and below our threshold. It also prevents us from counting the non-solid super-k-mers sharing their minimizer with a solid super-k-mer (indicating a possible sequencing error in the non-solid super-k-mer, whose k-mers shared with the solid super-k-mers should *not* be discarded).

As such, the second stage consists of correcting these issues by re-reading the input. Non-solid super-k-mers sharing their minimizer with solid super-k-mers which are substrings of hyper-k-mers already in the index, are added to the internal hash table. This also allows for the addition of the first and last super-k-mers of each read. After these two stages, the only k-mers that will be wrongly discarded are the ones that **A/** are present more than two times in the input but **B/** belong to a non-solid super-k-mer that does not share its minimizer with any of the solid super-k-mers. This super-k-mer-level threshold is slightly more aggressive than a k-mer-level threshold to filter unique k-mers. The effects of this heuristic are discussed in Section 14.4.2.

As our algorithm works in a streaming fashion, at any time, only the index has to be maintained in memory whereas reads are only loaded on-demand. This allows our implementation to have a light memory footprint (see Section 14.4), and to be parallelized (see Figure C.1).

KFC's algorithm is given in Algorithm 14.1. For clarity, we omit some details, such as how KFC handles subsequences in hyper-k-mer parts.

14.4. Experiments

In this section, we benchmark our proposed k-mer counter, KFC, based on hyper-k-mers and compare it against other counters: Jellyfish [MK11], KMC [DDG13; DKGD15; KDD17], FASTK [Mye23], Kaarme [DLS24] and Gerbil [ERM17]. Our implementation is written in Rust and is publicly available at https://github.com/lrobidou/KFC with accompanying test scripts at https://github.com/imartayan/KFC_experiments.

We report the datasets used in this study and their characteristics in Table C.1. The

experiments were performed on a common laptop computer (Dell Inc. Precision 7780 with 13th Gen Intel® Core™ i9-13950HX × 32 and 64 GB RAM).

14.4.1. Long reads whole genome sequencing datasets

We first evaluate all tools at ~100X coverage of the *E. coli* genome across three long-read technologies (ONT duplex, ONT simplex, HiFi) and k-mer sizes up to 1021, reporting memory and runtime in Figure 14.7. All competitors degrade as k grows: KMC3 and FastK quickly approach the available RAM with sharply increasing runtime, Gerbil is faster and lighter but capped at $k \leq 512$, Jellyfish trades runtime for memory, and Kaarme keeps memory nearly constant at the cost of much slower runs and frequent timeouts. KFC is the only tool whose memory and runtime both decrease with k, taking the lead in both metrics from around $k = 200$.

We confirmed these trends on two larger HiFi metagenomic benchmarks. Figure C.2 reports on a Zymo Biomics community sample of 21 strains across 17 species, with 5 Gb sampled from accession `SRR13128014`. Figure C.3 reports on a human gut metagenome used as assembly benchmark in [FCPL22; Ben+24] with estimated genome content of 700-900 Mb, where we pooled 15 Gb from accessions `SRR15275210-SRR15275213`. Tool behavior across coverages and on the full gut dataset is unchanged, with details in Figure C.4 and Figure C.5.

To check that KFC's advantage doesn't come from the super-k-mer filtering heuristic alone, we repeated the experiment without filtering (Figure C.6): Jellyfish is unaffected, FastK and KMC degrade further in time and memory, while KFC remains competitive even against the tools that still filter. The same picture extends to a pangenome regime: on one thousand *S. enterica* complete genomes (Section C.6), KFC keeps the lead in memory across all k-mer sizes and becomes the fastest tool past $k = 500$.

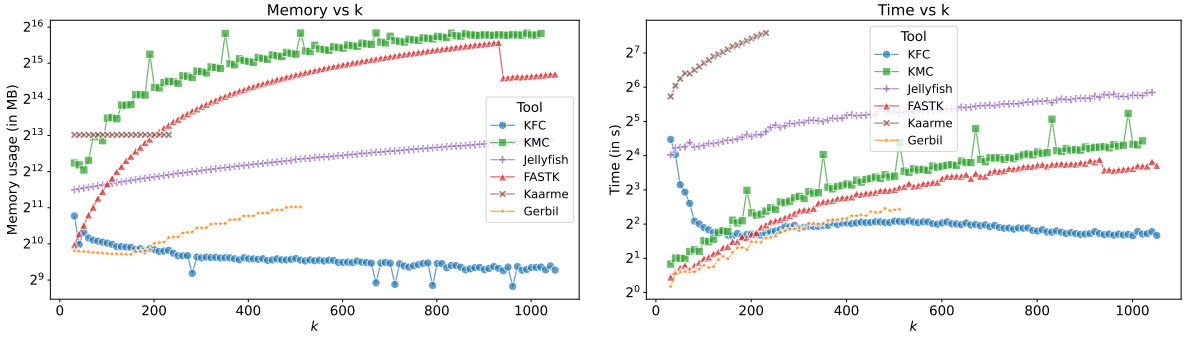
14.4.2. Super-k-mer threshold heuristic

We evaluate the super-k-mer abundance threshold by measuring how many k-mers are dropped at each threshold (Figure 14.8). The proportion of lost k-mers decreases exponentially with the threshold, and increases at lower coverage, larger k-mer sizes, and higher error rates, mirroring classical k-mer abundance filtering. To check that the lost k-mers are erroneous rather than genomic, we count the genomic k-mers absent from both KMC and KFC outputs across thresholds and k-mer sizes (Figure 14.9). KFC drops a few more genomic k-mers than KMC at low thresholds, but both methods converge as the threshold rises; on a dataset with a HiFi error rate (0.1%), KMC and KFC outputs match exactly. The super-k-mer-level threshold is therefore slightly more aggressive than a k-mer-level threshold but conservative enough to preserve accuracy.

14.5. Conclusion

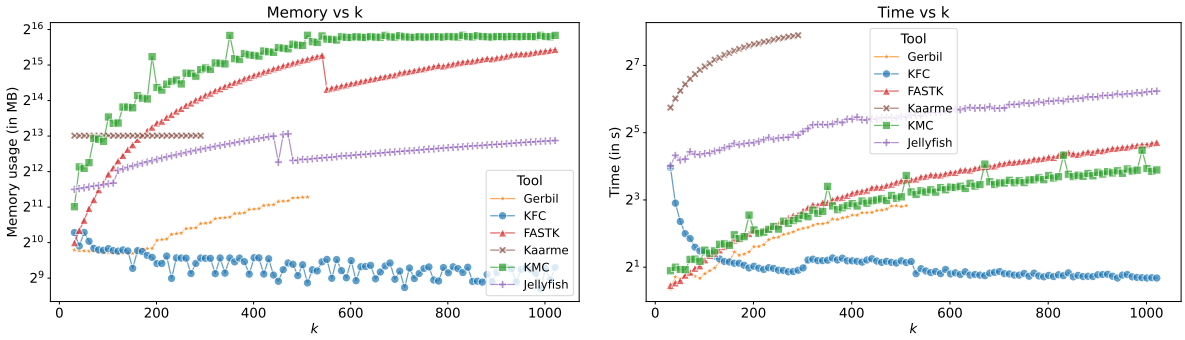
This chapter introduced hyper-k-mers as a streaming, partitioned alternative to super-k-mers for large k-mer sizes. By sharing each $k-1$ base overlap between consecutive super-k-mers rather than duplicating it, hyper-k-mers asymptotically reach 4 bits per nucleotide, two thirds of the super-k-mer cost. This matches the closed-syncmer asymptote while hyper-k-mers remain a stand-alone k-mer-set representation. We translated this representation to a practical k-mer counter, KFC, which becomes both the fastest and the most memory-efficient tool past $k \approx 200$.

This compactness comes at two structural costs relative to super-k-mers. Each hyper-k-mer is now a pair of pointers into a shared pool of left and right parts rather than a contiguous string, so access carries one extra indirection per side along with the corresponding bookkeeping. The representation is also no longer self-contained at the k-mer level: a given k-mer can be covered by multiple hyper-k-mers when the sequence is revisited, so applications that require a unique representation must deduplicate explicitly. This last point parallels the limitation



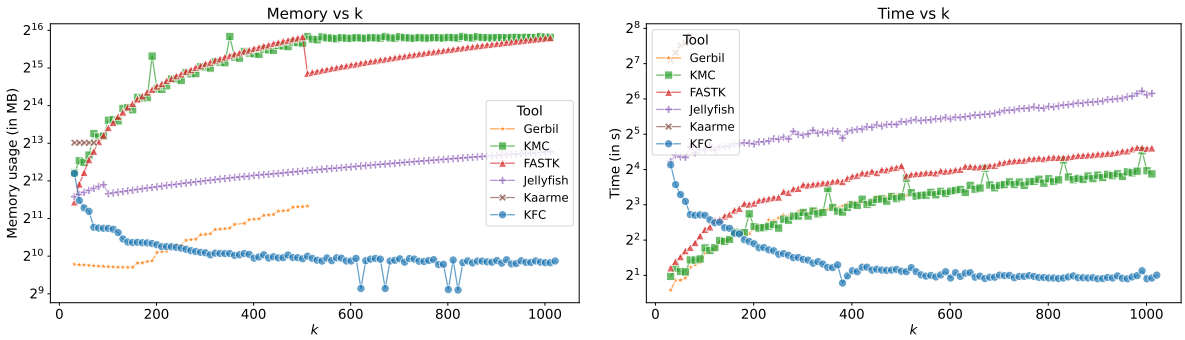
(a) Memory usage (b) Running time

Figure 14.4.: ONT Duplex *E. coli* datasets (`SRX23975767` and `SRX23975758`) at approximately 100X depth



(a) Memory usage (b) Running time

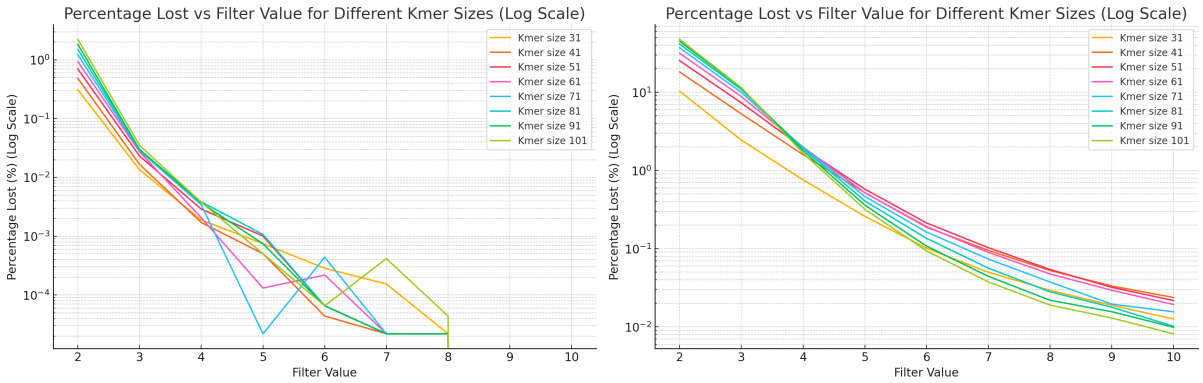
Figure 14.5.: HiFi *E. coli* dataset (`SRR11434954`) downsampled at 100X depth



(a) Memory usage (b) Running time

Figure 14.6.: ONT Simplex *E. coli* dataset (`SRR28370668`) downsampled at 100X coverage

Figure 14.7.: Comparison of k-mer counting benchmarks on different *E. coli* datasets, with unique k-mer filtering. Each subfigure shows the memory usage and timing plots for different sequencing technologies.



(a) HiFi dataset (b) ONT dataset

Figure 14.8.: Percentage of k-mers lost by removing unique super-k-mers, as a function of the abundance threshold and k-mer size. Left: simulated HiFi dataset with 100× coverage and 0.1% error rate. Right: simulated ONT dataset with 100× coverage and 1% error rate.

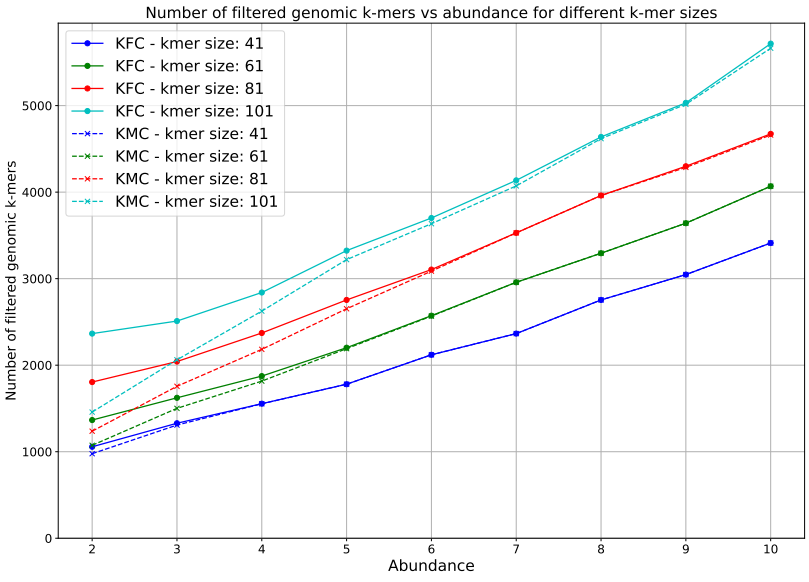


Figure 14.9.: Amount of filtered genomic k-mers by KFC and KMC according to the abundance threshold and k-mer size, on a simulated 100× ONT *E. coli* dataset with 1% error rate.

that the super-k-mer-level dictionary of Chapter 13 faces, and resolving it cleanly for either representation remains open work.

Compared to the static unitig-based representations of Chapter 2, which approach the 2-bit lower bound but require a separate counting pass and dissolve the per-minimizer partitioning, hyper-k-mers preserve both the streaming construction and the bucket structure that downstream indexes rely on for cache-friendly access and load balancing. The cost is the explicit deduplication step, and hyper-k-mers therefore sit structurally between super-k-mers and static SPSS representations.

Several directions remain open. The asymptotic 4 bits per nucleotide of hyper-k-mers is still twice the lower bound of 2 bits, and finding a representation that achieves this lower bound while keeping streaming construction is an open question. At the other end of the k-mer size spectrum, hyper-k-mers do not scale well to small k, where the smaller minimizer size leaves many distinct hyper-k-mer contexts under each minimizer bucket and bucket scans dominate the cost. Lengths below 16 can be handled by a plain hash table on k-mers themselves, but the range 17 to 63 is not well served by either approach and is a natural target for future work on dynamic linear representations. Finally, the current KFC algorithm discards k-mers belonging to non-solid super-k-mers that share no minimizer with a solid super-k-mer, as discussed in Section 14.3. This heuristic is highly efficient in practice, but handling the corner case without losing performance would close the small accuracy gap with k-mer-level filtering.

Algorithm 14.1: KFC's two-pass k-mer counting algorithm using hyper-k-mers.

```
function CountHyperkmer(read_set ∈ (Σ⁺)⁺; k, m, t ∈ ℕ with m ≤ k)
    sk_count ← empty multimap                    // maps minimizer to (hash(sk), count(sk))
    hk_count ← empty multimap                    // maps minimizer to (id_left, id_right, count(hk))
    hk_parts ← empty vector of bitpacked strings                    // 2 bits per character
    for read in read_set do                                         // First pass, can be parallelized
        for (prev_sk, sk, next_sk) in read do
            mini ← minimizer of sk
            if (hash(sk), c) ∈ sk_count[mini] then
                c ← c + 1
                increment the corresponding entry in sk_count[mini]
            else
                c ← 1
                add (hash(sk), 1) to sk_count[mini]
            if c = t then                                           // sk just became solid: insert its hyper-k-mer
                if IsSolid(sk_count, prev_sk, t) then // if previous is solid, reuse its right part
                    id_left ← id of right part of prev_sk in hk_count
                else
                    id_left ← |hk_parts|
                    append left part of sk to hk_parts
                if IsSolid(sk_count, next_sk, t) then       // if next is solid, reuse its left part
                    id_right ← id of left part of next_sk in hk_count
                else
                    id_right ← |hk_parts|
                    append right part of sk to hk_parts
                add (id_left, id_right, c) to hk_count[mini]
            else if c > t then                                      // sk already indexed: update count
                (id_left, id_right, exact) ← LookupAndUpdate(hk_count, hk_parts, sk)
                if not exact then
                    add (id_left, id_right, 1) to hk_count[mini]
    for read in read_set do                                         // Second pass, can be parallelized
        for sk in read do // handle non-solid super-k-mers sharing a minimizer with a solid
        one
            mini ← minimizer of sk
            if first or last super-k-mer then
                (id_left, id_right, exact) ← LookupAndUpdate(hk_count, hk_parts, sk)
                if not exact then
                    if id_left = −1 then
                        id_left ← |hk_parts|
                        append left part of sk to hk_parts
                    if id_right = −1 then
                        id_right ← |hk_parts|
                        append right part of sk to hk_parts
                    add (id_left, id_right, 1) to hk_count[mini]
            else if not IsSolid(sk_count, sk, t) then
                LookupAndUpdate(hk_count, hk_parts, sk)
    return sk_count, hk_count, hk_parts
```

Algorithm 14.2: KFC's auxiliary functions.

```
function IsSolid(sk_count, sk, t)          // checks if a super-k-mer has reached threshold t
    mini ← minimizer of sk
    for (h, c) ∈ sk_count[mini] do
        if h = hash(sk) then
            return c ≥ t
    return false

function LookupAndUpdate(hk_count, hk_parts, sk)
    // search if a super-k-mer is indexed in the hyper-k-mer index
    // return the position of the left and right hyper-k-mer parts that include it
    // and a boolean indicating if the super-k-mer is indexed
    // if the super-k-mer is indexed, the count of its entry is increased
    mini ← minimizer of sk
    id_left, id_right ← −1, −1                          // -1: sentinel for "not found"
    for (l, r, c) ∈ hk_count[mini] do
        if (hk_parts[l], hk_parts[r]) = sk then
            c ← c + 1
            return (l, r, true)
        if sk starts with hk_parts[l] then
            id_left ← l
        if sk ends with hk_parts[r] then
            id_right ← r
    return (id_left, id_right, false)
```

Second discussion

This part started with a simple question: can the representation of a k-mer set itself be a source of performance gains? The answer that emerged is yes, and a fairly broad one. Choosing a good unit of representation (a k-mer, a super-k-mer, a hyper-k-mer, or a necklace) shapes nearly everything downstream, from how compactly the set can be stored to how well streaming queries behave on a real CPU, whether dynamic updates are possible at all, and how easily two indexes can be combined. This is in some ways a continuation of the first part, where I argued that algorithm and implementation should be designed together. Here the same idea moves up one level. The data structure that holds the k-mer set is part of the algorithm, and choosing it well makes the rest of the pipeline simpler.

CBL, presented in Chapter 11, came out of this perspective at the k-mer level. By representing each k-mer through its necklace, consecutive k-mers naturally share long prefixes, and a quotiented layout turns this redundancy into both compression and fast streaming access. This layout also made it easy to support exact set operations in Chapter 12, something that very few k-mer indexes offer at all. Brisk, in Chapter 13, changes the indexing unit to whole super-k-mers, with the interleaved transform making binary search inside a bucket viable. KFC, in Chapter 14, goes further by sharing the $k-1$ overlap between consecutive super-k-mers, reaching 4 bits / k-mer while keeping streaming construction. Looking at these three together, my impression is that current k-mer indexes are already well established when it comes to space usage and single-query speed, as the comparison in Chapter 10 makes clear, and that the next frontier lies in widening the range of operations they support natively.

Dynamic updates and set operations are the two that I find the most compelling. Pangenomes and large public databases keep growing, and most workflows would benefit from being able to incrementally add new genomes to an existing index, or to combine indexes through union, intersection and difference without rebuilding from scratch. These operations also act as compact filters. Intersection isolates the shared core of a collection, while difference removes contaminants or extracts what is specific to a sample. CBL takes a small step in this direction, but I am convinced there is more to do, especially on the super-k-mer side where deduplication during streaming insertion is still an open problem, as discussed at the end of Chapter 13. A clean solution there would let super-k-mer dictionaries keep their streaming insertion without losing the inherent deduplication that k-mer-level structures get for free.

A direction I find promising is a closer convergence with full-text indexes. I chose not to cover them in this thesis, but they face closely related concerns and have made significant practical progress recently, with structures such as Movi [Zak+24; ZBGL26] pushing the performance of matching queries. The SBWT stands out in that landscape by bridging full-text approaches and k-mer methods, and recent work by Alanko et al. [ADMP26] even implements set operations directly on top of it. There seems to be a lot to learn between the two communities, and I expect more exchanges in the coming years.

Closer to the application side, I see a real opportunity in importing ideas from database query engines, as I outlined at the end of Chapter 12. Sort-merge joins, hash joins, partition pruning, exponential search on skewed inputs, SIMD-accelerated intersection of sorted lists, all of these have direct counterparts in k-mer set operations and have been refined for decades in the database literature. Vizitig [DPM25] is one concrete example of what becomes possible once a k-mer index is treated as a queryable database rather than a fixed pipeline component, and I expect this line of work to grow.

Multi-k approaches also strike me as an underexplored direction in the current literature. Even

though varying k along the analysis is already standard practice in assembly, where short k-mers help resolve low-coverage regions and longer ones disambiguate repeats, k-mer set structures are almost always built for a single fixed value of k. I think there is room for indexes that natively support several values of k, or that can answer queries at a finer granularity than the indexed one without rebuilding. The SBWT already supports querying any m-mer with $m \leq k$, which is a nice starting point.

Finally, the range of relevant k-mer sizes is itself evolving. Long-read technologies now reach error rates low enough that k-mers in the hundreds become tractable, as already exploited by KFC, and pangenomic comparisons benefit from this range. This shift makes the asymptotic behavior of representations matter more, and explains why moving from super-k-mers to hyper-k-mers pays off so visibly.

Part IV.

Sampling k-mers to lower memory & complexity

He who controls the space controls the universe.

—not Frank Herbert

Part I and Part II addressed throughput and representation of k-mer-based genomic pipelines. This part finally asks how much of the data can be discarded without losing the answers we care about during analysis. The observation driving this part is that many bioinformatics queries, such as similarity estimation, read mapping or large-scale search, do not require every k-mer. A carefully chosen subset, small enough to fit in memory and fast enough to compare, is often sufficient. The central challenge is therefore to characterize how small that subset can be, control its composition, and build the data structures that exploit it. Chapter 15 surveys the landscape of low-density minimizer schemes and shows the theoretical lower bound on how few k-mers any window-based sampling strategy must select. Chapter 16 combines multiple independent hash functions to push this boundary further. Chapter 17 applies these ideas at a larger granularity, sketching entire super-k-mers rather than individual k-mers to enable similarity queries at reduced memory and sublinear complexity.

15. Background on low-density minimizers

As discussed in Chapter 4, the *density* of a minimizer scheme drives its space efficiency: lowering the density leads to longer runs of the same minimizer and longer super-k-mers, which results in a direct improvement in applications such as k-mer counting [KDD17; MRSL25], DBG compaction [CLM16; KKDP22; CT23] or large-scale indexes [MKL21; Pib22]. However, we've also seen that the widely used *random minimizers* achieve a density of $2/(w+1)$, well above the $1/w$ lower bound implied by the window guarantee. This observation raises two main questions: how can we design new schemes with a lower density? And what is the lowest density we can hope to reach? In this chapter, we briefly cover some of the recent advances on these two questions, and a more detailed survey can be found in [Gro25a].

15.1. Universal hitting sets and decycling sets

15.1.1. Universal hitting sets

One of the first directions explored to lower the density emerged from *universal hitting sets* [Ore+16; Ore+17]. A universal hitting set, or UHS, is simply a set U of k-mers such that every window of w k-mers contains at least one element from U. From a DBG perspective (see Chapter 2 for some reminders), designing a UHS is equivalent to selecting a subset of nodes such that every path of length w is guaranteed to contain at least one of them.

Unlike minimizers, UHS are *context-free* in the sense that a selected k-mer does not depend on its surrounding context. We can however derive a minimizer scheme compatible with a UHS U by using an order that ranks all k-mers in U before all others. Since every window contains an element from U and the order favors these, we know that every selected minimizer would belong to U. The density of such a scheme is upper-bounded by $|U|/\sigma^k$ [MDK18], thus designing a smaller UHS directly translates to a lower density. Unfortunately, Orenstein et al. [Ore+16] proved that finding a minimum UHS is NP-hard by reducing it from the Hitting Set problem [Kar09], so we have to rely on heuristics to find small UHS instead.

15.1.2. Decycling sets

A popular heuristic to approximate a good UHS is to start from a *minimum decycling set* (MDS). The idea behind a decycling set is that, instead of covering every path of length w, we cover every cycle of the DBG. Luckily, building a minimum decycling set is much easier than a minimum UHS: Mykkeltveit [Myk72] constructed it by embedding each k-mer x into the complex plane as

$$z = \sum_{i=0}^{k-1} x_i \cdot \omega_k^i$$

where ω_k is a primitive k-th root of unity. We then select the k-mers whose embedding z corresponds to the first clockwise rotation with positive imaginary part, i.e. $\pi - 2\pi/k \leq \arg(z) < \pi$.

DOCKS [Ore+16; Ore+17] first suggested to greedily extend this MDS into a UHS for a specific window size w by repeatedly adding the k-mer contained in the largest number of uncovered windows, until all windows are hit. However, the exponential search space limited this approach to $k \leq 14$ in practice.

Pellow et al. [Pel+23] proposed a simpler and more scalable approach by building a minimizer scheme directly from this MDS: within any window, prefer k-mers in the MDS and break ties randomly. Checking MDS membership can be done by simply computing the corresponding embedding or with a precomputed lookup table, and the scheme extends to arbitrary k. Pellow et al. [Pel+23] also introduced *double decycling sets* as a variant that prioritizes a symmetric MDS by selecting k-mers with $-2\pi/k \leq \arg(z) < 0$, and falls back to Mykkeltveit's MDS otherwise. This approach further reduced density, especially for $k < w$ where it significantly outperforms DOCKS and similar UHS-based constructions. A plot of the density of double decycling sets, along with other schemes, is available in Figure 15.1.

15.2. (Open-closed) mod-minimizers

15.2.1. Mod-minimizers

Groot Koerkamp and Pibiri [GP24] took a different approach to improve the density of minimizer schemes by using a technique called *modulo sampling*. Their idea is to first select the position of a minimizer of length $t = k \bmod w$ using a random order, and then map it back to a k-mer position by applying a modulo w. The resulting scheme, named *mod-minimizers* is detailed in Algorithm 15.1. Note that when $k < w$, $t = k$ and this scheme is perfectly equivalent to random minimizers with no density benefit. In practice, t is often forced to be $\geq r$ by setting $t = r + ((k - r) \bmod w)$ for some small lower bound r[1], which avoids pathological cases such as selecting t-mers of length 1.

Algorithm 15.1: Pseudocode for mod-minimizers.

```
function ModMinimizers(w, k)
    t ← k mod w
    i ← 0
    for j in SlidingMinPos(w + k − t, t) do        // window of w + k − t t-mers
        yield i + ((j − i) mod w)                  // smallest integer i' ≥ i such that i' ≡ j mod w
        i ← i + 1
```

Groot Koerkamp and Pibiri [GP24] proved that the density of the mod-minimizers is given by

$$d = \frac{2 + \frac{k-t}{w}}{w + k - t + 1} + o(1/(w + k - 1))$$

which converges to $1/w$ as $k \to \infty$ with w fixed, leading to an asymptotically optimal scheme. A plot of the resulting density is available in Figure 15.1.

The scheme has the benefit of being very simple and efficient: it only requires computing a sliding minimum (which we discussed in Chapter 8) over short t-mers and a single modulo afterwards, making it a near drop-in replacement for random minimizers. Moreover, the modulo operation, which is usually quite expensive, can be accelerated using a "fast mod" [LKK19] since w is fixed. It should be noted, however, that this optimization does not apply as well on SIMD lanes of fixed size which cannot be widened. Practical experiments with SSHash [Pib22] showed that replacing random minimizers by mod-minimizers resulted in a 15% space gain for $w = 11$ and $k = 21$, without degrading the speed [GP24].

15.2.2. Open-closed mod-minimizers

One limitation of mod-minimizers is that they do not offer any improvement over random minimizers when $k < w$. To address this regime, Groot Koerkamp et al. [GLP25] introduced

[1] Using $r = 4$ seems to be a good empirical choice.

the *open-closed minimizers*, building on the notion of open and closed syncmers (presented in Chapter 4). Their idea is to first look for the smallest open syncmer in the window; if none exists, fall back to the smallest closed syncmer, and default to the globally smallest k-mer if neither is present. While the density of this scheme has no known closed form, it is similar to double decycling sets for $k < w$ with a small shift due to the inner s parameter of syncmers, and improves over them for $w \leq k \leq 2w$.

Groot Koerkamp et al. [GLP25] then proposed to combine this new scheme with the modulo sampling technique of mod-minimizers, thus resulting in *open-closed mod-minimizers*: t-mers are sampled using the open-closed minimizer as the inner scheme, and their position is mapped back to a k-mer modulo w. This merges the density gains of both approaches and achieves low density across the full range of k, as illustrated in Figure 15.1.

15.3. Lower bounds for forward local schemes

While the density of any scheme with a window guarantee can be trivially lower bounded by $1/w$, all of the improved schemes discussed previously remained quite far from it for small values of k, suggesting that a density of $1/w$ could simply be unreachable in this region.

Marçais et al. [MDK18] established a first non-trivial lower bound for *forward local schemes*, that is, schemes that select a k-mer only based on the content of the current window (*local*) and have an increasing sequence of selected positions (*forward*). This bound was then tightened by Groot Koerkamp and Pibiri [GP24], and later refined as a near-tight lower bound by Kille et al. [Kil+24]: any forward local scheme must satisfy

$$d \geq \frac{\left\lceil \frac{w+k}{w} \right\rceil}{w+k}$$

The key insight behind this result is to consider cyclic strings of length exactly $w+k$: on such a cycle, the window guarantee forces at least $\lceil (w+k)/w \rceil$ positions to be sampled, so the density per cycle is at least $\lceil (w+k)/w \rceil / (w+k)$. Since the density of any forward scheme equals its average density over all such cycles, the bound follows.

Moreover, Kille et al. [Kil+24] noted that this bound peaks at values $k \equiv 1 \mod w$ and can be smoothed into a monotone version since the optimal density decreases with k[2]:

$$d \geq \max\left(\frac{\left\lceil \frac{w+k}{w} \right\rceil}{w+k}, \frac{\left\lceil \frac{w+k'}{w} \right\rceil}{w+k'} \right)$$

where $k' = k + ((1-k) \bmod w)$ is the smallest integer $\geq k$ such that $k' \equiv 1 \mod w$. Figure 15.1 illustrates this lower bound along with the density of the schemes presented above.

15.4. Other variants

> **Note**
>
> As these approaches were published recently, they were not included in the following chapters.

[2] A scheme for k can always be turned into a scheme for $k' > k$ by ignoring the last $k' - k$ characters of each window.

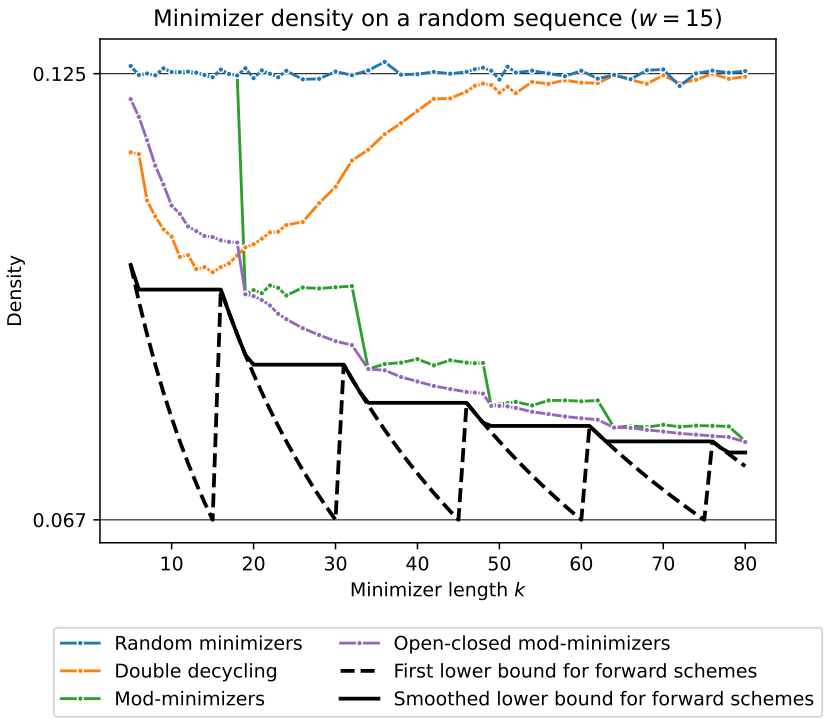


Figure 15.1.: Plot of the density of multiple minimizer schemes, along with the lower bound for forward local schemes from [Kil+24].

15.4.1. Finimizers

Finimizers [ABP25] are variable-length minimizers that ensure that the frequency of each selected k-mer stays below a given threshold. By avoiding very frequent k-mers, they reduce biases in downstream analyses, at the cost of a non-uniform k-mer length.

15.4.2. GreedyMini

GreedyMini [Gol+25] constructs a minimizer scheme through a greedy brute-force search. It iteratively selects the k-mer whose designation as the overall minimum would least often place it first or last in a context, thereby minimizing the number of charged contexts and hence the density. The resulting schemes achieve density very close to the lower bound and are optimal for some small parameter values when $k \equiv 1 \mod w$. However, this approach is not scalable for $k > 15$ as the construction time increases exponentially with k.

15.4.3. 10-minimizers

10-minimizers [STO26] are defined by an order that systematically prioritizes k-mers starting with the binary pattern `10` (named 10-k-mers), the idea being that they tend to be spaced further apart because the pattern cannot overlap itself. This translates into an expected density approximately $2/(w+2)$, slightly lower than the $2/(w+1)$ of random minimizers. Shur et al. [STO26] then refine this construction into *spacers* by ranking 10-k-mers to maximize the gap to the next selected position, achieving a competitive density.

15.4.4. Anti-lexicographic SUS-anchors

SUS-anchors [Gro26] are a scheme designed for $k = 1$, or *selection* scheme, that samples the start position of the smallest unique substring (SUS) in the window, i.e. the shortest substring

that occurs only once within the window and is smallest according to a given order. This scheme can be combined with an *anti-lexicographic* order, which compares the first character normally but reverses the order of all subsequent ones, to avoid over-sampling positions in long runs of the same character and push anchors further apart. Empirically, anti-lexicographic sus-anchors achieve density within 1% of the lower bound for selection schemes on the DNA alphabet.

15.5. Toward multi-hash schemes

All schemes surveyed in this chapter share a common design: they improve density by choosing which ordering to impose on k-mers, so that the minimum of each window falls as infrequently as possible. In the next chapter we explore a different strategy: we ask whether using multiple independent hash functions simultaneously can push density below what a single scheme can achieve.

16. Multiminimizers

> **Note**
>
> This chapter is adapted from Ingels et al. (2025), accepted to RECOMB 2026 and to be published.
> Antoine Limasset and I came up with the original idea. Florian Ingels proposed the formalization, and we did the proofs together. Lucas Robidou wrote most of the code, based on algorithms that we designed together. I also wrote the experiments script for the density analysis. Camille Marchet helped us writing the paper.

As surveyed in Chapter 15, recent progress on low-density minimizer schemes has brought practice very close to the forward local scheme lower bound, suggesting diminishing returns from further improvements in that direction. This chapter starts from this observation and asks a simple question: what if, instead of committing to a single minimizer scheme, we allow choosing among multiple independent candidates? The idea mirrors the *power of two choices* in load balancing [ABKU94; Mit01], where picking the less loaded of two random bins significantly reduces maximum load, and we ask whether the same strategy can further reduce minimizer density.

16.1. A link between density and the expected distance between selected positions

We start by formalizing the link between density and distances between consecutive selected positions, under minimal assumptions on these distances. Specializing to random minimizers, we argue that the standard modeling assumption is unsatisfactory and propose a more realistic one that coincides with the classical regime for large minimizers, verifying empirically that it satisfies our minimal condition.

> **Different notations**
>
> We reuse the notations from Section 4.3: the minimizer size is denoted m while k denotes the number of bases in a window, i.e. $k = w + m - 1$.

We follow the notations of Chapter 4 and denote $\mathcal{O}_m$ for a total order on m-mers and $\mathsf{rank}_{\mathcal{O}_m} : \Sigma^m \to [\![1, \sigma^m]\!]$ the corresponding rank function. In practice, $\mathcal{O}_m$ is induced by a random hash function with $U = 2^{64}$, which we treat as collision-free [PSL23]. We denote by $\mathcal{S}_f(S)$ the set of selected indices of a string S under a scheme f, and by $d(S) = |\mathcal{S}_f(S)|/(|S| - k + 1)$ its particular density. The expected density d is the limit of $\mathbb{E}[d(S)]$ over random strings of growing length. The definitions below are stated in a non-canonical context, though they extend easily. The practical experiments are based on canonical k-mers, but our implementations cover both contexts.

Consider the expected number of selected positions in a window of size w, equal to $d \cdot w$ by definition. Intuitively, this quantity is also equal to w divided by the expected distance between two selected consecutive positions, which we denote here by μ. Indeed, if we select a position every μ bases, we expect to choose w/μ positions on average in a window. Hence,

with $dw \approx w/\mu$, we expect $d \approx \frac{1}{\mu}$. This intuitive link has already been noticed in the literature [SWA03; ZKM21; MRSL25], but never, to the best of our knowledge, explicitly formalized as such, nor proved for any scheme. The purpose of this section is to formally establish this observation as a theorem, namely Theorem 16.1, under minimal assumptions.

The sequence S is represented by a vector of integers $\mathbf{R} = R_1, R_2, ...$, where $R_i = \mathsf{rank}_{\mathcal{O}_m}(S[i : i+m]) \in [\![1, \sigma^m]\!]$ is the rank of the i-th m-mer of S, according to the order $\mathcal{O}_m$. A second integer vector, $\mathbf{P} = P_1, P_2, ...$, describes the positions of the selected minimizers when sliding a window of size w, i.e.

$$P_i = \min\{i \leq j \leq i + w - 1 : R_j = \min(R_i, \dots, R_{i+w-1})\}.$$

Note that $\mathbf{P}$ can still be defined no matter how minimizers are chosen, or more generally, a position, in a window. Since consecutive windows may choose the same position as a minimizer, consecutive values of P_i might be equal. It leads us to define the subsequence $\mathbf{P}^*$ of *distinct* values, i.e. $P^*_1 = P_1$, P^*_2 is the first P_i value distinct from P^*_1, P^*_3 is the first P_i value distinct from P^*_1 and P^*_2, and so on. In other words, $\mathbf{P}^*$ is the sequence corresponding to the set $\mathcal{S}_f(S)$ – i.e. the set of selected indices. Finally, we define $\Delta = \Delta_1, \Delta_2, ...$ as the sequence of distances between selected positions, i.e. $\Delta_1 = P^*_1$, and $\Delta_i = P^*_i - P^*_{i-1}$ for $i \geq 2$. An example is provided in Figure 16.1.

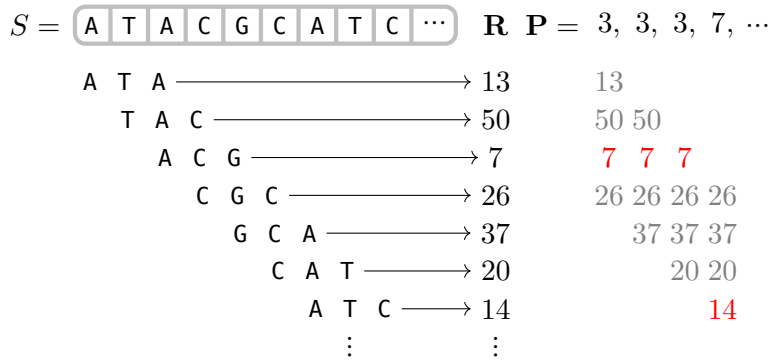


Figure 16.1.: An example of sequence S, represented as a sequence $\mathbf{R}$ of integers (here, using lexicographical order), and the sequence $\mathbf{P}$ of minimizers positions, with $m = 3$ and $w = 4$. We have $\mathbf{P}^* = 3, 7, ...$ and $\Delta = 3, 4, ...$

We put ourselves in the context where S is a random sequence, so that $\mathbf{R}, \mathbf{P}, \mathbf{P}^*$ and Δ can be considered as vectors of random variables. Our goal is to establish a link between d and the expected distance between selected positions, i.e., between d and the $\mathbb{E}[\Delta_i]$'s. We obtain the following result.

Theorem 16.1 (Density-distance equivalence)**.** Let $\mu > 0$. Consider $\varepsilon_i = \mathbb{E}[\Delta_i] - \mu$ as some realization of an underlying random variable ε with $\mathbb{E}[\varepsilon] = 0$. Then, we have $d \cdot \mu = 1$.

Proof. The proof is deferred to Section D.1. □

Theorem 16.1 states that if we assume that the distances between consecutive selected positions of minimizers are somehow equally distributed, then their expected value is exactly $1/d$. This assumption is the minimum viable hypothesis for establishing the result (since μ must be well defined), and it does not depend on the process by which the R_i's values are obtained, nor how the positions of the minimizers $P_1, P_2, ...$ are selected in successive windows.

At this point, we have not made any assumptions on the random vectors $\mathbf{R}, \mathbf{P}$ and $\mathbf{P}^*$. It is not trivial to formally verify whether the assumption regarding the $\mathbb{E}[\Delta_i]$'s is verified or not, as the variables are most likely not to be independent nor identically distributed. However, in

practice, we can verify that the assumption holds. In the remainder of this section, we investigate how the assumption from Theorem 16.1 concerning the Δ_i's holds for random minimizers. The standard hypothesis that has been made in the literature regarding their choice is the following:

Hypothesis 16.1. Every m-mer in a window of length w has an equal probability of $1/w$ of being the smallest m-mer.

As discussed in [Mar+17], while "not strictly true in practice", this hypothesis is reasonable and reflects reality accurately "when using a randomized ordering". However, the formulation is unfortunate because it does not take into account the dependency between windows. If we consider a window $(R_i, \dots, R_{i+w-1})$ outside of its context, then the hypothesis can be reformulated as $P_i \sim \mathcal{U}([\![1, w]\!])$. Unfortunately, since the same minimizer will likely be selected by several successive windows, P_i cannot be considered independent, nor even identically distributed (since $P_i = P_{i-1}$ with a non-zero probability). Here, we propose starting from a more elementary hypothesis concerning the R_i's. We then study, empirically and theoretically, the behavior of the derived variables $\mathbf{P}$, $\mathbf{P}^*$ and Δ, to assess the gap between Hypothesis 16.1 and reality, and to test the expectation hypothesis on the Δ_i's. Our working assumption is the following.

Hypothesis 16.2. The random variables $R_1, R_2, \dots$ are i.i.d., uniformly distributed over $[\![1, \sigma^m]\!]$.

Remark 16.1. Assuming a perfect hash function when computing the random order, our assumption states that the probability of collision between R_i's is $1/\sigma^m$, which is consistent with [ZKM21, Lemma S7], that establishes that two distinct m-mers are identical in a random sequence with probability $1/\sigma^m$.

It is not too difficult to establish the law of $P_1 = P_1^* = \Delta_1$ from Hypothesis 16.2:

Proposition 16.1 (Uniform distribution of P_1)**.** Under Hypothesis 16.2, $\forall 1 \leq i \leq w$,

$$\mathbb{P}(P_1 = i) = \frac{1}{w} + \left(\frac{w-i}{w-1} - \frac{1}{2}\right) \cdot \frac{1}{\sigma^m} + O\left(\frac{1}{\sigma^{2m}}\right).$$

It follows that $P_1 \xrightarrow{a.s.} \mathcal{U}([\![1, w]\!])$ when $m \to \infty$.

Proof. The proof is deferred to Section D.2. □

In Section D.3, we observe that the limit distribution is reached very quickly, for fairly small values of m ($m = 8$ in our case). This implies that for all values of m likely to be used in practice, Hypothesis 16.1 holds for P_1, and Δ_1 also follows a uniform distribution, meaning that $\mathbb{E}[\Delta_1] = \frac{w+1}{2}$. If, indeed, all subsequent Δ_i's have the same expectation, then, using Theorem 16.1, we retrieve the well-known density of random minimizers, $d = \frac{2}{w+1}$ [SWA03; Rob+04].

Unfortunately, formally establishing the law of further variables becomes rapidly intractable due to the dependencies between them. To complete the study of subsequent $\mathbb{E}[\Delta_i]$'s, we resort to Monte-Carlo simulations of our probabilistic model and obtain the results depicted in Section D.4. From our results, the assumption that $\mathbb{E}[\varepsilon] = 0$ is empirically verified (numerically, we get $\mathbb{E}[\varepsilon] \approx 0.0060$), and the $\mathbb{E}[\Delta_i]$'s are indeed distributed around $\mu = \frac{w+1}{2} = 5.5$ (using $w = 10$); see supplementary Figure D.2a. We obtain numerically a density of 0.1808, i.e. 0.55% of error compared to the theoretical value of $0.\overline{18}$. This empirical result highlights that Hypothesis 16.2 is indeed a proper assumption to work with, and also that the assumption of Theorem 16.1 is reasonable, since it is verified in practice.

16.2. Multiminimizers: trading time for space

We now introduce a practical family of meta schemes called *multiminimizers*, that assign a bounded set of candidate minimizers and choose among them to reduce the frequency of selected positions.

Considering Theorem 16.1, obtaining a low density scheme on a sequence S requires having a scheme that selects positions as far as possible from one another in S, while still covering all k-mers from S. In this regard, the best possible minimizer scheme would select m-mers at a distance of $k - m + 1$ bases from each other. However, without a way to retrieve which m-mer was chosen for a specific k-mer, querying a k-mer would require checking if any of its $k - m + 1$ m-mers was selected as its minimizer. The same problem arises when inserting a k-mer into a database: one must perform $k - m + 1$ checks to verify whether it is already present or not.

To reduce this prohibitive cost, we propose a solution based on super-k-mers. As defined in Section 4.3, a super-k-mer is a maximal sequence of consecutive k-mers sharing the same minimizer. Our technique starts by generating N distinct hash functions (e.g. using N distinct random seeds). Then, we use these N hash functions to generate N distinct random minimizer schemes. Each of these minimizer schemes yields a set of super-k-mers on the sequence S, each covering all k-mers of S. When iterating over the minimizers of a sequence, among the super-k-mers that cover the first k-mer, we select the one that ends the farthest in the sequence. Then, we repeat this selection at the end of each selected super-k-mer: among all the super-k-mers covering the first uncovered k-mer, we select the one that ends the farthest in the sequence (see Figure 16.2). We detail our method in Algorithm 16.1, running in $\mathcal{O}(N \cdot |S|)$ time and using $\mathcal{O}(N \cdot w)$ memory.

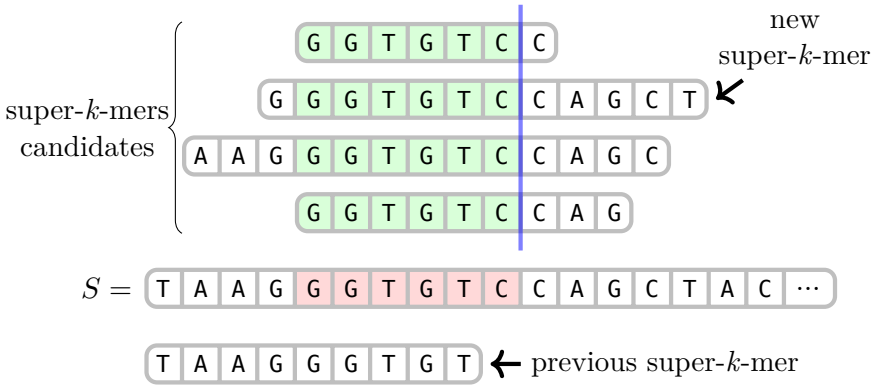


Figure 16.2.: Choice of a super-k-mer for $k = 6$ using a multiminimizer scheme. The first k-mer uncovered by the previous super-k-mer is highlighted in red. Among the $N = 4$ candidate super-k-mers covering this k-mer, the multiminimizer scheme selects the one ending the farthest in the sequence (i.e. farthest right of the blue line).

As mentioned in the introduction, a *local scheme* selects a m-mer in a k-mer solely based on the k-mer itself. This ensures covering all k-mers, or equivalently, all windows of $w = k - m + 1$ m-mers [MDK18]. Formally, a local scheme is a function $f : \Sigma^k \to [\![1, w]\!]$. Importantly, the local scheme is both used at construction (e.g. when partitioning k-mers based on their minimizer) and query (e.g. to determine in which bucket a k-mer might be present). For a given k-mer $x \in \Sigma^k$, the position $f(x)$ is uniquely determined.

While minimizers are local schemes, multiminimizers are not, as they differ drastically in their approach. At construction, we need to know when the previous super-k-mer ends, and where the candidate super-k-mers end, information that a k-mer alone cannot provide. In this regard, the multiminimizer scheme has both to "remember the past" to determine when to select a new minimizer, and "delve into the future" to choose which minimizer candidate to keep. At query, for a given k-mer, we have to compute and look up the N minimizer candidates to determine

Algorithm 16.1: MultiMinimizers algorithm.

function MultiMinimizers($T, w, m, \mathcal{S}_0, \dots, \mathcal{S}_{N-1}$) *// T: text; $\mathcal{S}_0, \dots, \mathcal{S}_{N-1}$: N minimizer schemes.*
  Initialize `buf` of size $N \times w$ and `run` of size N
  $k \leftarrow w + m - 1$
  $i^* \leftarrow -w$ *// Starting position of the current super-k-mer.*
  $j^* \leftarrow 0$ *// i^* mod w.*
  $h^* \leftarrow 0$ *// Selected minimizer scheme.*
  `run`$[h^*] \leftarrow 0$
  for $0 \leq i < |T| - k + 1$ **do**
    if `run`$[h^*] = 0$ **then** *// The minimizer of the selected scheme has changed.*
      $j \leftarrow i \bmod w$
      $r \leftarrow i - i^*$ *// Number of outdated values in the buffer.*
      for $0 \leq h < N$ **do**
        for $i^* + w \leq p < \min(i^* + w + r,\ |T| - k + 1)$ **do**
          `buf`$[h][p \bmod w] \leftarrow \mathcal{S}_h(T[p : p + k])$
        `run`$[h] \leftarrow 1$
        while `run`$[h] < w$ **and** `buf`$[h][(j +$ `run`$[h]) \bmod w] =$ `buf`$[h][j]$ **do**
          `run`$[h] \leftarrow$ `run`$[h] + 1$
      $i^*, j^* \leftarrow i, j$
      $h^* \leftarrow \arg\max$ `run`
    `run`$[h^*] \leftarrow$ `run`$[h^*] - 1$
    yield `buf`$[h^*][j^*]$

whether it has already been seen, since the broader context(s) in which it might have been seen are not known.

Ultimately, the multiminimizer scheme does not behave like a local scheme, either at construction or at query time. It is advantageous as it implies that its density is not subject to the lower bound of local schemes by Kille et al. [Kil+24], although the $1/w$ bound still applies. As showed in upcoming Section 16.4.1, the multiminimizer scheme does indeed beat this lower bound, and approach a density of $1/w$ as N increases. This gain in density comes at the price of (bounded) increased query time, as discussed above.

Finally, note that the multiminimizer construction is not limited to random minimizer schemes, but could be used with any combination of N distinct local schemes, whether by varying parameters or using concurrent schemes; in that sense, multiminimizers are a sort of *meta* scheme. To illustrate this point, in upcoming Section 16.4, we investigate two implementations of the multiminimizer construction, one with random minimizers, and the other with open-closed mod-minimizers [GLP25].

16.3. On deduplicated density

It appears the standard density possesses a "twin" concept, that we call *deduplicated density* (which we denote by d^*), and that we define as the fraction of distinct minimizers needed to cover all k-mers of a set of sequences (or, equivalently, a set of k-mers), instead of the fraction of positions selected along a sequence. The deduplicated density does not appear to have been studied before, even if it is implicitly linked to filtering tasks, such as the ones tackled by Needle [Dar+22].

Definition 16.1 (Minimizer Cover)**.** Let $\mathcal{X} = \{x_1, x_2, \dots\}$ be a set of (distinct) k-mers. The *minimizer cover* of $\mathcal{X}$, according to some minimizer scheme f, that we denote by $\text{MinCover}_f(\mathcal{X})$,

is defined as $\mathrm{MinCover}_f(\mathcal{X}) = \{\min_f(x) : x \in \mathcal{X}\}$, where $\min_f(x)$ designates the minimizer of x according to f.

Analogously to standard density, we can define a *particular* deduplicated density $d^*(\mathcal{X})$ and obtain simple bounds, mimicking the ones we already have on standard particular density:

$$\frac{1}{|\mathcal{X}|} \leq d^*(\mathcal{X}) = \frac{|\mathrm{MinCover}_f(\mathcal{X})|}{|\mathcal{X}|} \leq 1.$$

Standard density is computed as the limit of the expected particular density $\mathbb{E}[d(S)]$ on a random sequence S when $|S| \to \infty$, for fixed values of k and m. However, one cannot have an arbitrarily large set $\mathcal{X}$ since $|\mathcal{X}| \leq \sigma^k$; therefore, we propose to define the deduplicated density simply as the expected particular density of a random set of k-mers $\mathcal{X}$, without any notion of limit.

Definition 16.2 (Deduplicated density)**.** The (expected) deduplicated density of a minimizer scheme f is formally defined as

$$d^* = \mathbb{E}_{\mathcal{X}}[d^*(\mathcal{X})] = \frac{1}{2^{\sigma^k} - 1} \sum_{\mathcal{X} \in \mathcal{P}(\Sigma^k) \setminus \emptyset} d^*(\mathcal{X})$$

where $\mathcal{P}(\Sigma^k)$ designates the powerset of Σ^k.

Noticing that $1/|\mathcal{X}| \geq 1/\sigma^k$, we obtain the following trivial bounds for d^*.

Lemma 16.1 (Bounds on deduplicated density)**.** For any minimizer scheme, we have $1/\sigma^k \leq d^* \leq 1$.

Then, we have the following result.

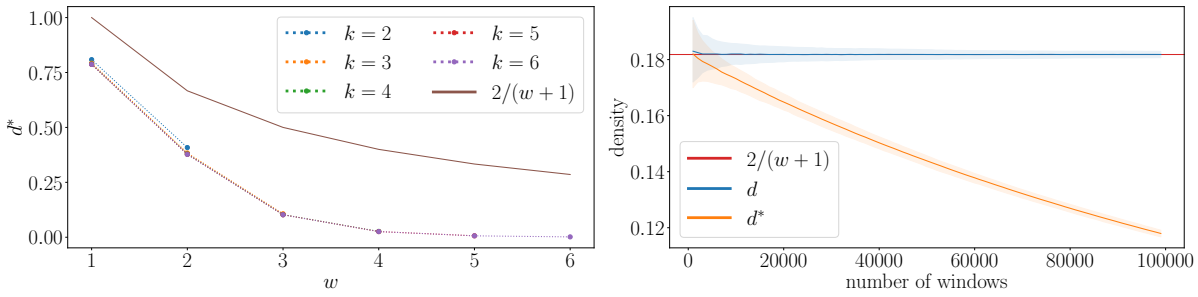
Proposition 16.2 (Deduplicated density of random minimizers)**.** Under Hypothesis 16.2, the deduplicated density of the random minimizer scheme is given by the following, where $w = k - m + 1$:

$$d^* = \frac{1}{2^{\sigma^k} - 1} \sum_{n=1}^{\sigma^k} \binom{\sigma^k}{n} \frac{\sigma^m}{n} \left[1 - \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left[1 - \left(\frac{r}{\sigma^m} \right)^w + \left(\frac{r-1}{\sigma^m} \right)^w \right]^n \right].$$

Proof. The proof is deferred to Section D.5. □

The dependence on σ^k makes it intractable to compute d^* when k is large. We nonetheless computed the first values, shown in Figure 16.3 (left); one can see that d^* seems to be driven by the value of w rather than k, hence we conjecture that d^* might only depend on w. Besides, it seems that $d^* < d$ holds. To access higher values, we resorted to Monte-Carlo simulations, following the same probabilistic model as before, using Hypothesis 16.2. We observed how the standard density d and the deduplicated density d^* evolved over the course of parsing a random sequence, as shown in Figure 16.3 (right). While both densities seem to be equal for the first few thousand windows, they rapidly diverge. We looked into the data and observed that the number of distinct k-mers and distinct windows observed are equal (it is not surprising since the probability of collision of k-mers is negligible), so the difference between the densities is entirely explained by the discrepancy between the number of selected positions and the number of distinct minimizers. While those values are equal at first, soon we obtain repeated minimizers[1], hence the drop in d^*.

[1]While the *positions* of minimizers are somehow uniformly distributed within a window, as seen in Section 16.1, it is not the case of their *values*, which are the ones that repeat and therefore lead to a decrease in d^*. Indeed, the values of the minimizers are distributed as $\min(R_1, \ldots, R_w)$ where R_i are i.i.d. uniform variables, and therefore are strongly biased towards small values.



(a) Theoretical values of d^* for $2 \leq k \leq 6$ as a function of $1 \leq w \leq k$, obtained with the formula of Proposition 16.2.

(b) Monte-Carlo simulations of random minimizers under Hypothesis 16.2. We generated $N_{\text{simu}} = 10^3$ random sequences $\mathbf{R}$ of size $|\mathbf{R}| = 10^5$, with $m = 8$ and $w = 10$. 95% of values across all simulations are within the colored areas, the solid lines being the medians.

Figure 16.3.: (left) Theoretical values of d^* for small k. (right) Evolution of d and d^* over the course of parsing a random sequence.

There are obviously many more aspects of deduplicated density to be studied, but they are beyond the scope of this work. To conclude this section, we propose to consider applying multiminimizers to deduplicated density.

Definition 16.3 (Multiminimizer Cover)**.** Let $\mathcal{O}_m^1, \dots, \mathcal{O}_m^N$ be N orders on m-mers, and let $\mathcal{X} = \{x_1, x_2, \dots\}$ be a set of (distinct) k-mers. A set $\mathcal{Y} = \{y_1, y_2, \dots\}$ of m-mers is said to be a *multiminimizer cover* of $\mathcal{X}$ if and only if, for any $x \in \mathcal{X}$, there exists $y \in \mathcal{Y}$ and $1 \leq i \leq N$, so that $\min_i(x) = y$, where $\min_i(x)$ designates the smallest m-mer of x according to the order $\mathcal{O}_m^i$. In other words, for each k-mer $x \in \mathcal{X}$, at least one of its N minimizer candidates belongs to $\mathcal{Y}$.

The natural extension of the deduplicated density in this context would be $d^* = \frac{|\mathcal{Y}|}{|\mathcal{X}|}$, with the same trivial bounds as before. Contrary to $\text{MinCover}_f(\mathcal{X})$, we have here a choice of which candidate to choose as a minimizer, and therefore a chance at minimizing d^* by making the appropriate choices. Therefore, this leads to considering the following problem.

Problem 16.1 (MultiMinCover)**.** Provided a set $\mathcal{X}$ of k-mers and N orders on m-mers $\mathcal{O}_m^1, \dots, \mathcal{O}_m^N$, find a set $\mathcal{Y}$ of m-mers that is a multiminimizer cover of $\mathcal{X}$ and of minimum cardinality $|\mathcal{Y}|$.

This problem is closely related to the one of **MinimumHittingSet**: provided an integer k, and a set $\mathcal{X}$ of k-mers, find a set $\mathcal{Y}$ of m-mers (called a hitting set) such that each k-mer of $\mathcal{X}$ contains at least one m-mer from $\mathcal{Y}$, and of minimum cardinality. This problem is known to be NP-hard [Ore+16; Ore+17]. Here the problem is slightly different, as we do not ask the elements of $\mathcal{Y}$ to simply appear in the k-mers of $\mathcal{X}$, but also to be their minimizers for at least one of the orders. This problem also turns out to be difficult, as the following result shows.

Theorem 16.2 (NP-completeness of MultiMinCover)**.** MultiMinCover is NP-complete (for $\sigma \geq 3$ and $N \geq 2$).

Proof. The proof is deferred to Section D.6. □

Since minimizing $|\mathcal{Y}|$ is difficult, we must resort to heuristics. Furthermore, when parsing long sequences, it would be prohibitively expensive to keep track of all the k-mers encountered and their candidate minimizers before starting to build the multiminimizer cover. For this reason, we have developed a heuristic that makes decisions throughout the parsing of a sequence, based solely on the local context, that is presented in Section 16.4.3.

16.4. Results

In this section, we evaluate the performance of two implementations of the multiminimizer technique: an iterator over random-minimizers-based multiminimizers, and a proof of concept of multiminimizers using open-closed mod-minimizers [GLP25]. Both are written in the Rust programming language, and freely available at github.com/lrobidou/multiminimizers. We propose results for canonical k-mers, but our implementation also supports non-canonical k-mers.

Density results are provided in Section 16.4.1, whereas the space usage is investigated in Section 16.4.2 and the conservation of sampled multiminimizers is studied in Section D.8. Regarding running time, note that the multiminimizer scheme must compute all candidate minimizers, so iteration over multiminimizers is linear in the number of hash functions. We measured the time required to iterate over a random sequence and confirmed that our implementation scales linearly with the number of hash functions, while remaining very fast as shown in Section D.9. Finally, we provide some results in Section 16.4.3 on multiminimizers applied to filtering approaches.

16.4.1. Density

As discussed in Section 16.2, the multiminimizer construction aims at reducing the density of any local scheme. We expect to reduce the density when increasing the number of hash functions. The intuition is that increasing the number of super-k-mer candidates increases the chance of finding a super-k-mer that ends further than the others, thus increasing the distance between consecutive selected positions.

To show this density reduction, we generated a random sequence (5M bases) and computed the density of our scheme on it, for 1 to 8 hash functions, using $w = 15$ and varying m (thus k). We observed that the higher the number of hash functions, the lower the density, which allows us to reach densities lower than any other scheme (see Figure 16.4a). We are not aware of any prior iterator over minimizers with a density lower than the lower bound introduced by [Kil+24]. Figures for additional m and w values are available in Section D.7.

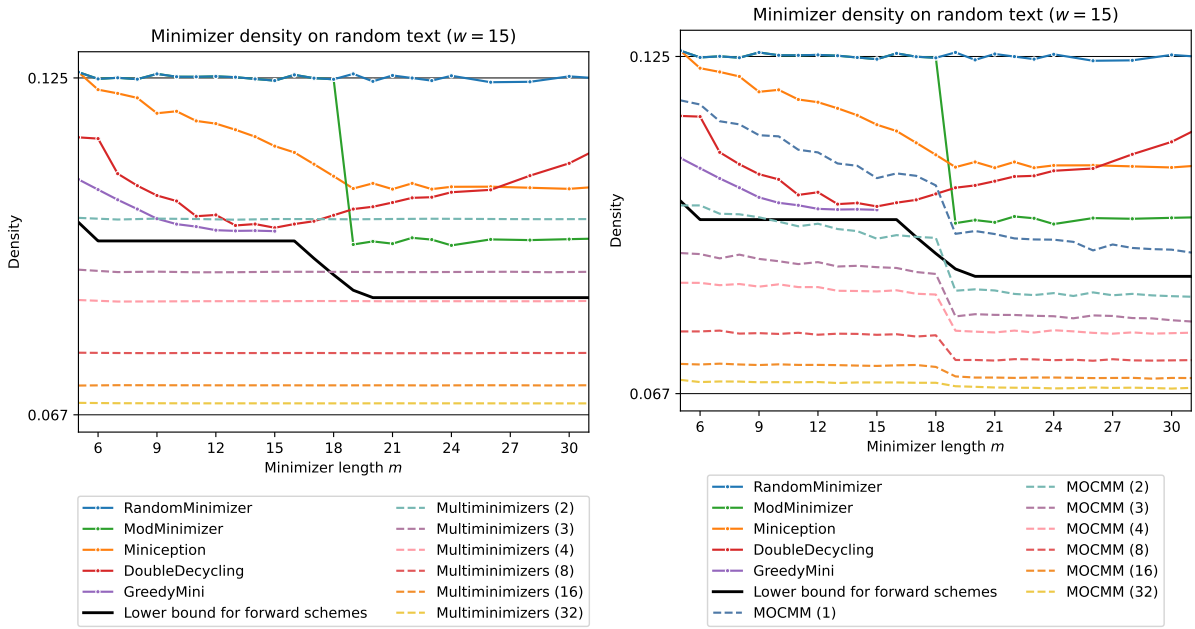
Although multiminimizers are not a local scheme, we have observed that the empirical density $\hat{d}$ and the empirical average distance between selected positions $\hat{\mu}$ were indeed related as $\hat{d} \approx 1/\hat{\mu}$, hence showing that Theorem 16.1 might apply beyond the scope of local schemes.

To show that the multiminimizer construction can be applied to other schemes than random minimizers, we applied it to the open-closed mod-minimizer [GLP25] to obtain a proof-of-concept of a multi open-closed mod-minimizer scheme (MOCMM for short). Figure 16.4b shows that this novel scheme achieves a lower density than the multiminimizers on a random sequence. At the time of writing, the library we used to compute the minimizer scheme does not support the open-closed mod-minimizer scheme. Therefore, we used a non-SIMD implementation, making it much slower. For this reason, we could only reasonably run it on a random sequence of 5×10^5 bases.

In both Figure 16.4a and Figure 16.4b, one can observe that the density approaches the theoretical lower bound $1/w$. This convergence seems intuitive since, when the number of hash functions becomes large enough, all m-mers of a k-mer are likely to be chosen as the minimizer candidate for at least one of the hash functions, and therefore the multiminimizer scheme tends to become optimal.

16.4.2. Super-k-mers and hyper-k-mers space usage

Since the space usage of a super-k-mer representation of a sequence is driven by the density of the underlying scheme [MRSL25, Thm. 1], we can translate a density reduction to a linear super-k-mer representation. Figure 16.5a shows the space usage of a super-k-mer representation of multiminimizers, whereas Figure 16.5b provides the same information for MOCMMs, both for multiple numbers of hash functions and k values.



(a) Multiminimizers.

(b) Multi open-closed mod-minimizers.

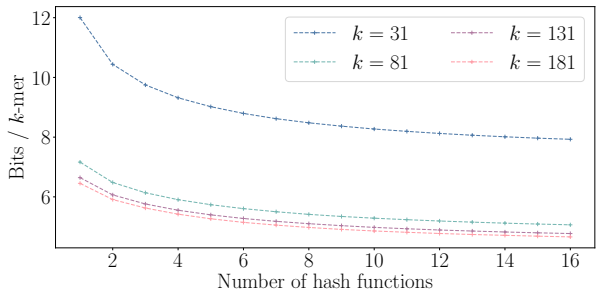
Figure 16.4.: Density of multiple schemes, compared to our (a) multiminimizer and (b) MOCMM schemes with up to 32 hash functions. Our schemes are able to reach densities less than the lower bound of [Kil+24]. `GreedyMini` is close to this bound, but its computation time is exponential in m (the next data point would take $\approx$ 62h to compute).

Ideally, to represent a sequence over an alphabet of size 4 (e.g., a DNA sequence S), one could use only 2 bits / k-mer in S. However, super-k-mers use as low as 6 bits / k-mer in S. hyper-k-mers [MRSL25] were recently introduced and tend toward 4 bits / k-mer in S when using a random-minimizer-based hyper-k-mer representation. This is still higher than the lower bound of 2 bits / k-mer, due to random minimizers having a density of $\frac{2}{w+1}$.

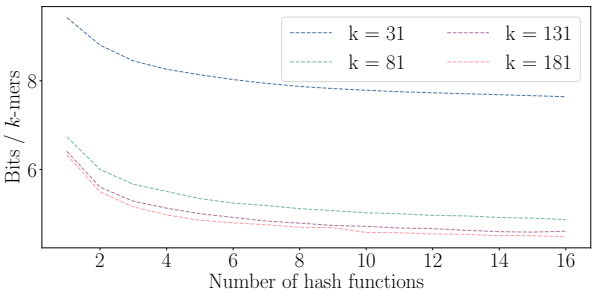
Since our multiminimizer scheme approaches a density of $\frac{1}{w}$, using a multiminimizer scheme could allow hyper-k-mers to tend toward 2 bits / k-mer in the input sequence. To test this hypothesis, we replaced the iterator over minimizers of KFC (a hyper-k-mer-based k-mer counter introduced in [MRSL25]) by our iterator over multiminimizers and report the result in Figure 16.5c. Note that KFC is optimized for large k values (e.g. > 60), so we tested our hypothesis in this range. Our results show that KFC converges to 2 bits / k-mer in S, which we believe to be the first streaming k-mer representation to do so.

16.4.3. Pin: a multiminimizer approach to Needle-like applications

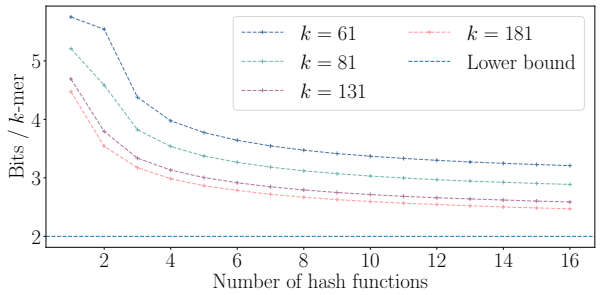
We developed a simple proof-of-concept minimizer index to demonstrate the utility of the multi-minimizers approach for Needle-like filtering stages, in a prototype dubbed Pin (github.com/Malfoy/Pin). Such an index stores a set of minimizers that collectively cover all k-mers of interest, enabling efficient filtering as an indexed minimizer is a necessary condition for an indexed k-mer, and with sufficiently large minimizers, passing the filter already implies substantial sequence similarity. Our prototype relies on exact hash tables and handles a single uncolored set; the same idea can be readily adapted to colored or approximate index structures. During construction, we test the presence of k-mers and then greedily select minimizers to cover any uncovered sequences. We evaluate the approach on the Human HiFi accession `SRR11292123` (~24 Gb) in Figure 16.6, and observe a smaller index at a moderate cost in build and query time. Merely switching from



(a) Super-k-mer representation + multiminimizers.



(b) Super-k-mer representation + MOCMM.



(c) Hyper-k-mer representation + multiminimizers.

Figure 16.5.: Number of bits / k-mer for different representations of a random read, with $m = 21$, depending on k and the number of hash functions used with the multiminimizer construction. Increasing the number of hash functions allows the hyper-k-mer representation (c) to converge to 2 bits / k-mer, the minimum required to represent a DNA sequence.

one to two hash functions yields an index about 20% smaller, with roughly a 20% increase in construction time and about 85% in query time.

16.5. Conclusion

Multiminimizers trade a bounded amount of query time for lower density. The construction is agnostic to the underlying scheme: we instantiated it with random minimizers and with open-closed mod-minimizers, and in both cases the density drops below the forward-scheme lower bound of [Kil+24] as the number of hash functions grows, approaching $1/w$. The gain carries over to downstream representations. Plugging our iterator into KFC brings hyper-k-mer storage close to 2 bits / k-mer, the first streaming k-mer representation to reach this regime as far as we know. The Pin prototype shows the same pattern for filtering indexes, where even $N = 2$ already cuts index size by about 20% at modest cost.

The chapter also developed two pieces of theory that stand on their own. The density-distance equivalence of Theorem 16.1 gives a way to reason about density under a single, minimal assumption on the expected spacing of selected positions, and it appears to hold empirically even outside the local-scheme setting where we proved it. The deduplicated density d^* emerged as a natural twin to standard density when the object of interest is a k-mer set rather than a sequence. We gave an analytical expression in the random-minimizer case, observed that it diverges from d on long sequences, and showed that minimizing it under the multiminimizer model is NP-complete.

Several questions remain open. The theoretical density of the multiminimizer scheme is still to be worked out, ideally as a function of N and of the densities of the base schemes; conservation properties also lack a formal treatment. The complexity of minimizing standard density globally

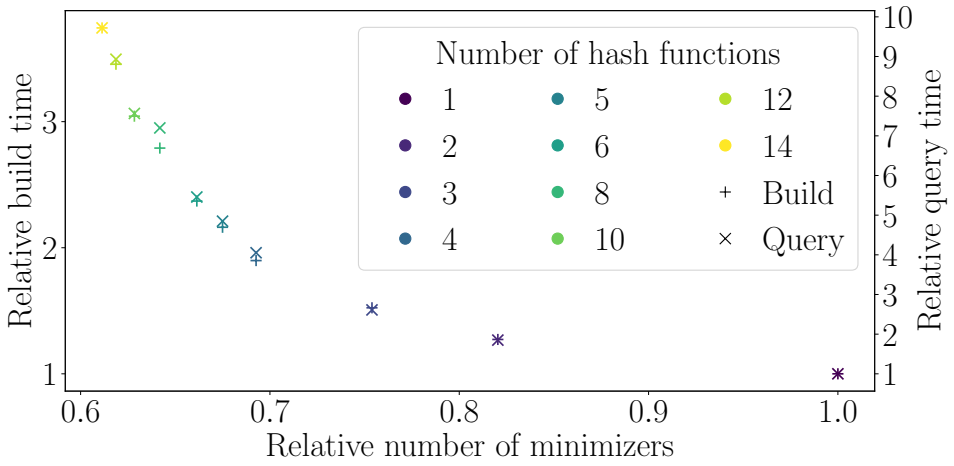


Figure 16.6.: Relative performances of the minimizer dictionary Pin using multiminimizers, comparing N hash functions vs $N = 1$. The x-axis depicts the relative index size (in terms of the number of minimizers), and the y-axis provides relative construction (left) and query (right) times. As an indicator, for $N = 1$, there are $\approx 850 \times 10^6$ minimizers, construction time is 234s, and query time is about 0.04s.

is still unknown, despite the NP-completeness result for its deduplicated counterpart. Extending deduplicated density to other scheme families, and simplifying its expression, would help clarify how the two notions relate. Better local or global heuristics for MultiMinCover seem worth pursuing.

On the practical side, the MOCMM proof of concept already shows that pairing multiminimizers with other low-density schemes is straightforward; a SIMD-friendly implementation is the natural next step. Replacing existing iterators with our Rust library is also a low-effort way to test whether the gains we saw on hyper-k-mer-based counting extend to other minimizer-based tools.

17. Sketching super-*k*-mers

> **Note**
>
> This chapter is adapted from Rouzé et al. (2023) and Rouzé et al. (2025), accepted to WABI 2023 and published in LIPIcs and Algorithms for Molecular Biology.
> Antoine Limasset and Camille Marchet came up with the original idea. Timothé Rouzé wrote most of the code and ran all the experiments. I proposed the notion of fractional hitting sets, derived the bounds, wrote all the proofs and implemented the decycling sets optimization.

As discussed in Chapter 3, FracMinHash [Irb+22] is a practical method for large-scale containment and Jaccard estimation of k-mer sets: its sketches scale linearly with input, avoiding the fixed-budget limitations of tools like Mash [Ond+16], and can handle documents of widely differing sizes. The storage strategy as implemented in sourmash [Pie+19] treats each selected k-mer as an opaque 32-bit hash, without exploiting the overlapping structure that consecutive k-mers naturally share. For collections at the scale of NCBI RefSeq, which contains over 1.2 million bacterial genomes totalling more than 5 Tbp, this per-element overhead becomes a bottleneck.

The key observation we build on is that consecutive k-mers share overlaps and naturally group into super-k-mers (see Section 4.3), making it possible to represent a set of selected k-mers more compactly than by storing individual hashes. Universal Hitting Sets (discussed in Chapter 15) aim to cover all k-mers with a low-density scheme; however, our application does not require complete coverage of the k-mer space. We therefore introduce *Fractional Hitting Sets* (FHS) that select a near-uniform fraction of the k-mer space, study the achievable density as a function of that fraction, and show that the resulting super-k-mers are particularly amenable to compact encoding. We implement this scheme in a tool called SuperSampler, and evaluate it against sourmash [Pie+19].

17.1. Preliminaries

Throughout this chapter we fix a DNA alphabet $\Sigma = \{A, C, G, T\}$ with $\sigma = |\Sigma| = 4$, and consider two input sets of sequences S_A and S_B (in practice, read sets or assembled genomes), with A and B denoting their corresponding k-mer sets.

> **Different notations**
>
> We reuse the notations from Section 4.3: the minimizer size is denoted m while k denotes the number of bases in a window, i.e. $k = w + m - 1$.

Definition 17.1 (maximal super-k-mer)**.** Let s ($|s| \geq k$) be a string and v a super-k-mer of s. Let i_m be the first position of the minimizer on s. v is a maximal super-k-mer iff v starts at position $i_m + m - k$ in s and v ends at position $i_m + k - 1$ in s. It follows that v has a length of $2k - m$ and contains $w = k - m + 1$ k-mers.

Examples of regular and maximal super-k-mers are shown in Figure 17.1. As noted in [PSL23, theorem 3], the proportion of maximal super-k-mers approaches $\frac{1}{4}$ for large k.



Figure 17.1.: Super-k-mers extracted from a sequence for $k = 6, m = 3$. Minimizers are shown in pink, here we use the lexicographic order instead of hashing minimizers for the sake of simplicity. Super-k-mers 1 and 4 are maximal (they contain respectively k-mers {CTGAAA, TGAAAT, GAAATG, AAATGC} and {TGCACA, GCACAT, CACATT, ACATTT}), while 2 and 3 are not (and contain respectively k-mers {AATGCA} and {ATGCAC}).

As recalled in Section 4.2, random minimizers achieve a density of $2/(w+1)$, and we define the *density factor* as the density multiplied by $w+1$ (so random minimizers have a factor of 2). The question we explore here is whether relaxing full coverage to a fraction f of k-mers allows us to push the density factor below 1, and what the implications are for sketch compactness.

17.2. Fractional Hitting Sets

A *fractional hitting set* relaxes the UHS requirement by only requiring that a fraction f of windows are hit, rather than all of them.

Definition 17.2 (fractional hitting set or FHS)**.** Given $f \leq 1$, a set $\mathcal{F} \subseteq \Sigma^m$ is a Fractional Hitting Set (FHS) if a fraction at least f of the sequences of w consecutive m-mers have an element contained in $\mathcal{F}$.

To avoid selection bias, we use a random hash function that distributes m-mers uniformly over $[\![1, \sigma^m]\!]$ and select all m-mers whose hash falls below a fixed threshold. We call these *small minimizers*. Note that any method selecting a fraction of the minimizers hashes would be suitable here.

Definition 17.3 (small m-mer)**.** Given a fixed threshold $t \in [\![1, \sigma^m]\!]$, we say that a m-mer is small if its hash is below t. We denote by $\mathcal{S}$ the set of small m-mers, and $p = \frac{t}{\sigma^m}$ the probability that a m-mer is small.

From p we can derive the proportion of covered k-mers.

Proposition 17.1. Given $t \in [\![1, \sigma^m]\!]$ and $p = \frac{t}{\sigma^m}$, the expected fraction of k-mers with distinct m-mers containing a small m-mer is

$$f = 1 - (1-p)^w$$

where $w = k - m + 1$ and p is the probability that a m-mer is small.

Proof. Given a k-mer with w distinct m-mers $x_1, \ldots, x_w$,

$$\Pr(\forall i \in [\![1, w]\!], h(x_i) > t) = \prod_{i=1}^{w} \Pr(h(x_i) > t) = (1-p)^w$$

because the hashes of distinct m-mers are independent, so

$$f = \Pr\left(\min_{i \in [\![1,w]\!]} h(x_i) \leq t\right) = 1 - (1-p)^w$$

□

Note that this property is valid for k-mers with distinct m-mers; k-mers with duplicated m-mers (i.e. k-mers containing repetitions) have a lower coverage since they have fewer candidates for small minimizers. Fortunately, as shown in [ZKM20] (see lemma 9), the proportion of k-mers with duplicated m-mers is negligible for a sufficiently large m ($> (3+\varepsilon)\log_\sigma w$). Thus, our selection method is uniform for k-mers with distinct m-mers, and almost uniform in the entire k-mer space.

Conversely, if we want a given fraction f of the k-mers to be covered, the threshold should be chosen as

$$t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m \tag{17.1}$$

Related to this, let us define the subsampling rate as $s = \frac{1}{f}$. For instance, a desired subsampling rate of 1000 will give $f = \frac{1}{1000}$.

17.2.1. Density of small minimizers

We showed that selecting k-mers with a small minimizer induces an FHS. By considering the usual definition of density (from Definition 4.1) for this scheme (which covers a fraction f of all k-mers), we obtain the following bound (proven in Section E.2):

Theorem 17.1. Given $f \leq 1$ and $t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m$, assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers in a random sequence is upper bounded by

$$\frac{2f}{w+1} + o(1/w)$$

At first glance, the results may be surprising, as the density is smaller than the lower bound of $1/w$ for $f < 1/2$ and can approach zero. This is because some k-mers may not contain any small minimizers and are therefore not covered, and the proportion of such k-mers increases as f approaches 0. However, it is worth noting that this bound does match the $2/(w+1)$ density when $f = 1$ (i.e., when every k-mer is covered).

To obtain a more meaningful metric, we can compute the density on the fraction of the sequence that is covered, instead of the entire sequence. With this approach, we obtain the following theorem, which has been proven in the Section E.3:

Theorem 17.2 (restricted density)**.** If we restrict to sequences in which every k-mer contains a small minimizer, then given $f \leq 1$ and $t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m$, assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers is upper bounded by

$$2 \cdot \frac{f + (1-f)\ln(1-f)}{f^2(w+1)} + o(1/w)$$

Although less intuitive than the previous one, this result provides valuable insights into the density within the covered portion of the sequence. As shown in Figure 17.2a, the associated density factor ranges from 2 when $f = 1$ (consistent with existing results) to 1 when $f = 0$. Therefore, as f approaches 0, we can approach the optimal density.

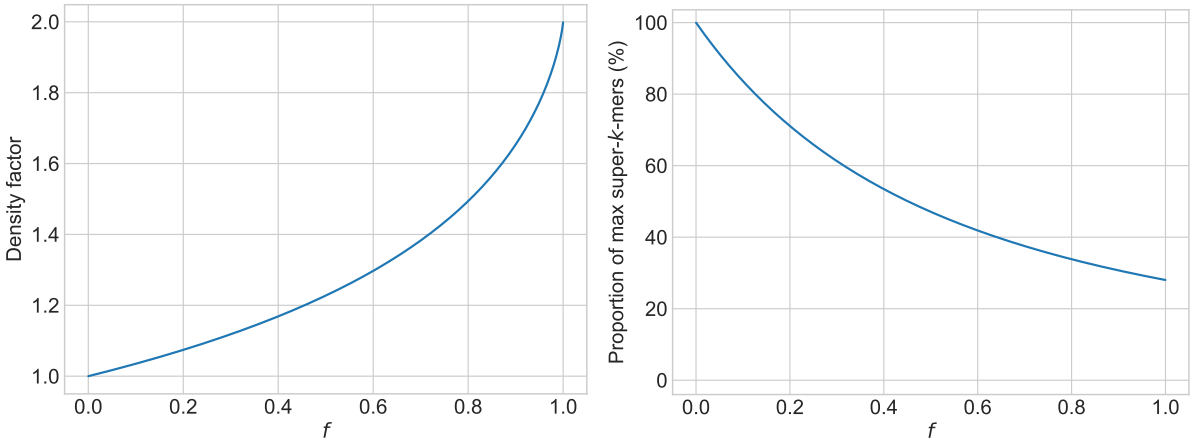
17.2.2. Proportion of maximal super-*k*-mers

Although measuring the density provides an overview of the average length of super-k-mers, it does not indicate how many of them are maximal (i.e., of length $2k-m$). The following result (proven in the Section E.4) answers this question:

Theorem 17.3. Given $f \leq 1$ and $t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m$, the average proportion of maximal super-k-mers (with respect to all super-k-mers) built from small minimizers in a random sequence is given by

$$\left[\left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]^2 + \frac{1 - f(1-2/w)}{1+f}$$

Note that this result generalizes theorem 2 from [PSL23], which corresponds to $f = 1$. As shown in Figure 17.2b, the proportion increases towards 100% as f approaches 0.



(a) Theoretical upper bound on the density factor from Theorem 17.2, lower is better.

(b) Average proportion of maximal super-k-mers built from small minimizers (for $k = 31, m = 15$), higher is better.

Figure 17.2.: Theoretical bounds on the density factor and the proportion of maximal super-k-mers depending on the fraction f of covered k-mers.

17.2.3. Improving the density of fractional hitting sets using UHS

This effect is more pronounced for smaller values of f. This observation raises a natural question: what is the lowest achievable density for a given f? Since universal hitting sets (UHS) with a density lower than 2 have been proposed for $f = 1$, it is possible that they may also improve the density for smaller f values by considering only the m-mers selected by the UHS as potential minimizers.

Theorem 17.4. Given a UHS $\mathcal{U}$ with density $d_{\mathcal{U}}$, $f \leq 1$ and $t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m$, assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers selected from $\mathcal{U}$ (that is, $\mathcal{S} \cap \mathcal{U}$) in a random sequence is upper bounded by

$$f \cdot d_{\mathcal{U}} + o(1/w)$$

The proof is given in Section E.5. Note that this result generalizes Theorem 17.1 since the UHS of minimizers selected using a random ordering has a density of $2/(w+1)$ [SWA03].

17.3. Sketching technique in SuperSampler

17.3.1. SuperSampler's sketch construction

Definition 17.4 (SuperSampler's sketch)**.** Given a sequence S, each super-k-mer whose minimizer's hash is lower or equal to a threshold t is selected, and all its surrounding k-mers are kept in the sketch as a super-k-mer. A SuperSampler sketch can therefore be represented as a super-k-mer set.

Sourmash stores each selected k-mer as a 32-bit hash, which introduces a small false-positive rate but keeps the encoding simple and sequence-agnostic. As illustrated in Figure 17.3, SuperSampler instead stores k-mers as super-k-mers: fingerprints are exact (no false matches,

shared k-mers can be retrieved), and overlapping k-mers within a super-k-mer share bases, reducing space.

When using random minimizers, super-k-mers contain $(k-m+1)/2$ k-mers on average [PSL23], giving a mean length of $(3k-m-1)/2$ bases. Each super-k-mer also needs a $\log_2(k-m+1)$-bit length header to delimit it, yielding a lower bound in bits / k-mer of

$$\frac{2\left(3k-m-1+\log_2(k-m+1)\right)}{k-m+1}$$

Using Equation 17.1 to set the fraction f lowers the density further, producing longer super-k-mers. At low f, almost all super-k-mers are maximal (Theorem 17.3), which removes the need for length headers (all have the same length $2k-m$) and gives the tighter bits / k-mer bound

$$\frac{2(2k-m)}{k-m+1}$$

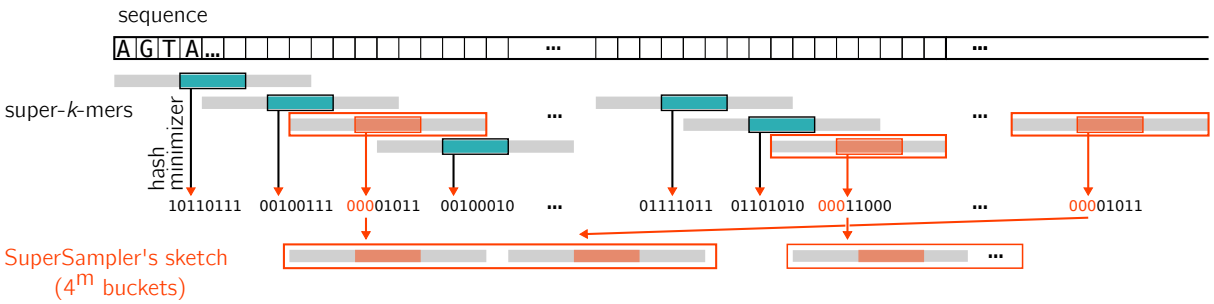


Figure 17.3.: SuperSampler's sketching strategy. In order to build sketches, SuperSampler computes super-k-mers over the input sequence. Fingerprints are associated with each super-k-mer by hashing their minimizers to an integer, hence an integer per super-k-mer. Super-k-mers associated to sufficiently low integers are kept in the sketch. Super-k-mers are put into partitions according to their minimizer.

17.3.2. Partitioned sketches

Since each super-k-mer is centered around a unique minimizer occurrence, super-k-mers naturally partition into groups sharing the same minimizer. SuperSampler stores each non-empty partition (a minimizer and its associated super-k-mers) independently. Within a partition, the minimizer sequence is stored once and omitted from all maximal super-k-mers, since its position within each maximal super-k-mer is known beforehand. This gives an even lower bits / k-mer ratio of

$$\frac{4(k-m)}{k-m+1}$$

Figure 17.4 shows the space cost of the three encodings (plain super-k-mer, maximal super-k-mer, partitioned super-k-mer) as a function of minimizer size, alongside measured performance on random sequences.

Partitioning also accelerates comparisons: matching k-mers between two sketches must share a minimizer, so only one partition needs to be in memory at a time.

17.3.3. Set comparisons

SuperSampler's sketch comparison produces an estimator for both Jaccard index $J(A,B) = |A \cap B|/|A \cup B|$ and containment index $C(A,B) = |A \cap B|/|A|$.

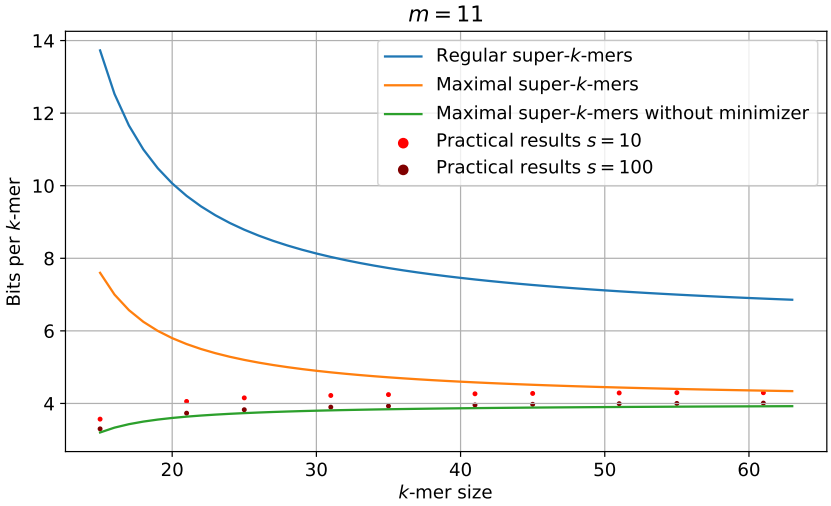


Figure 17.4.: Theoretical space cost of different encodings in bits / k-mer according to the k-mer size along with practical space usage of SuperSampler sketches on random sequences.

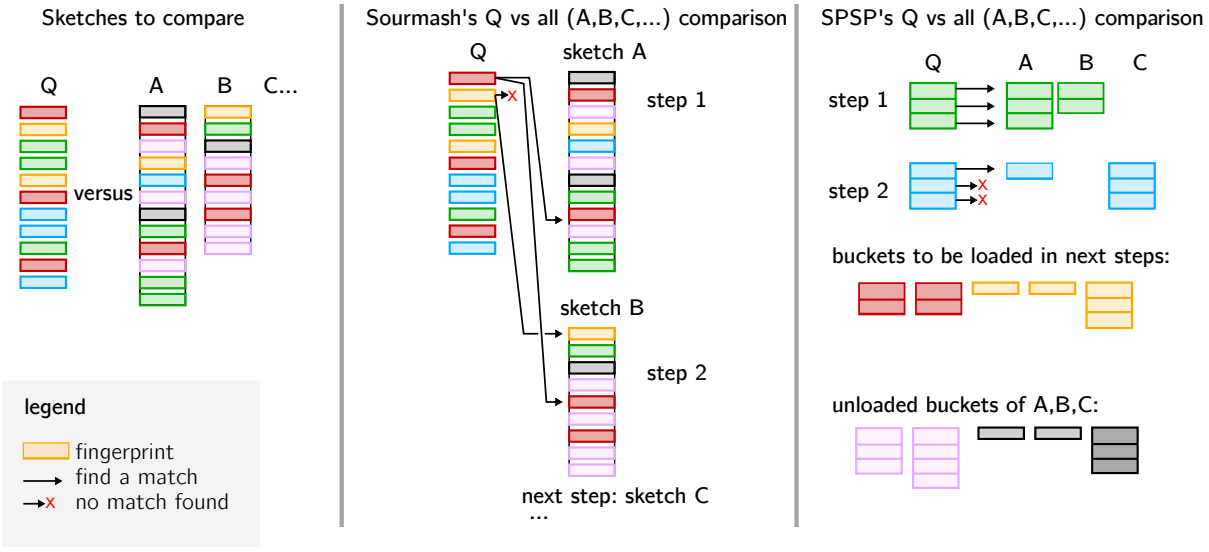


Figure 17.5.: How SuperSampler and sourmash perform their respective sketch comparison. Colored rectangles represent k-mers. Those sharing the same color are sharing a common minimizer. In SuperSampler sketches, k-mers sharing their minimizers are stored in the same partition. In this example, we discuss the comparison of one document against a collection, although other use cases can be inferred. SuperSampler is capable of skipping certain partitions that are not relevant to the query. By focusing on smaller sub-parts of the collection one at a time, SuperSampler effectively improves practical performance and reduces memory usage.

Unlike sourmash, which compares all fingerprints globally (Algorithm 17.1), SuperSampler iterates over minimizer partitions (Algorithm 17.2, Figure 17.5): partitions absent from the query are skipped entirely, and the remaining ones are processed independently. This is particularly beneficial for large collections, where most partitions are query-specific and can be skipped.

Algorithm 17.1: Sketch comparison in sourmash.

Let Q be a list of query files $Q_0..Q_i..Q_{s-1}$
Let T be a list of target files $T_0..T_i..T_{p-1}$
HQ is an array of s sets of fingerprints, one for each query file
HT is built similarly for each target
M is a matrix of size $s \times p$ filled with 0
for all $i \in HQ$ **do**
    for all $j \in HT$ **do**
        $M[i][j] = |HQ[i] \cap HT[j]|$ *// time linear in the size of the smallest set*

Algorithm 17.2: Sketch comparison in SuperSampler.

Let Q be a list of Query files $Q_0..Q_i..Q_{s-1}$
HQ is a hashtable of hashtables
HQ maps any minimizer in a file Q_i from Q to a hashtable
The second hashtable maps fingerprints from a given minimizer to a list of indexes of source files in $0..s-1$
Similarly, a hashtable of hashtables HT is built for each target dataset $T_0..T_{p-1}$
M is a matrix of size $s \times p$ filled with 0s
for all minimizer $m =$ key in HQ **do**
    if $m \in HT$ **then**
        for all fingerprint $f =$ key in $HQ[m]$ **do**
            if $f \in HT[m]$ **then** *// Partition $HT[m]$ is loaded only if necessary*
                for all index $dq \in HQ[m][f]$ **do**
                    for all index $dt \in HT[m][f]$ **do**
                        $M[dq][dt]$ += 1

17.4. Results

All experiments were run on a cluster node with an Intel Xeon Gold 6130 CPU (2.10 GHz) and 128 GB of DDR4 RAM.

17.4.1. Space efficiency of SuperSampler.

For $m = 15$, the first bound from Section 17.3.1 gives 9.2 bits / k-mer with $k = 31$ and 7.1 with $k = 63$, but in practice SuperSampler achieves approximately 6.5 and 5 bits / k-mer respectively as depicted in Figure E.2, thanks to the lower density of the FHS scheme. As shown in Figure 17.6, the density factor drops quickly with the subsampling rate: already below 1.04 at rate 100, and approaching 1 at rate 1000 (the sourmash default). Combining the FHS scheme with double decycling sets [Pel+23] gives an additional improvement, particularly noticeable for small k (Figure 17.7).

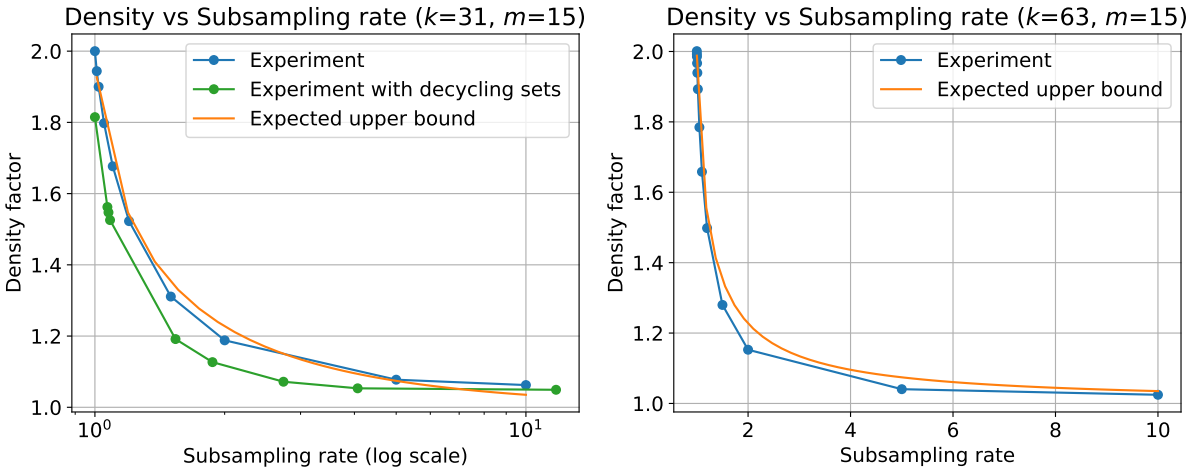


Figure 17.6.: Measured density factor compared to the model.

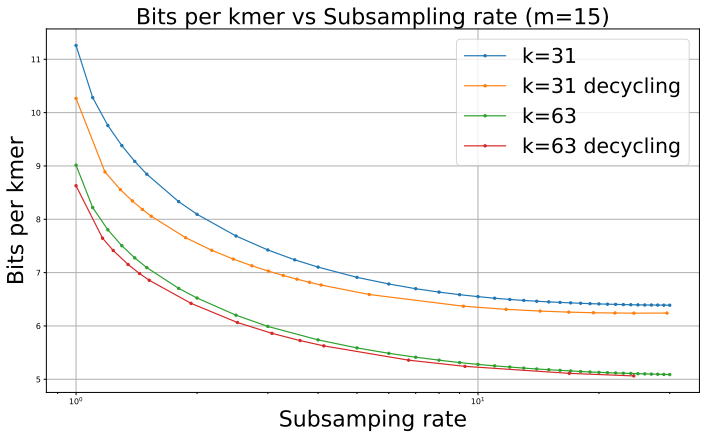


Figure 17.7.: Space cost in bits / k-mer according to the subsampling rate with and without using decycling sets (yellow and red lines).

17.4.2. Performance Comparison

We compared SuperSampler against sourmash on two datasets: 1024 Salmonella genomes (high mutual similarity) and 1024 RefSeq bacterial genomes (Jaccard similarity near 0). All-vs-all comparisons were timed and profiled with Snakemake, and exact Jaccard and containment values from Simka [Ben+16] serve as ground truth for precision estimates.

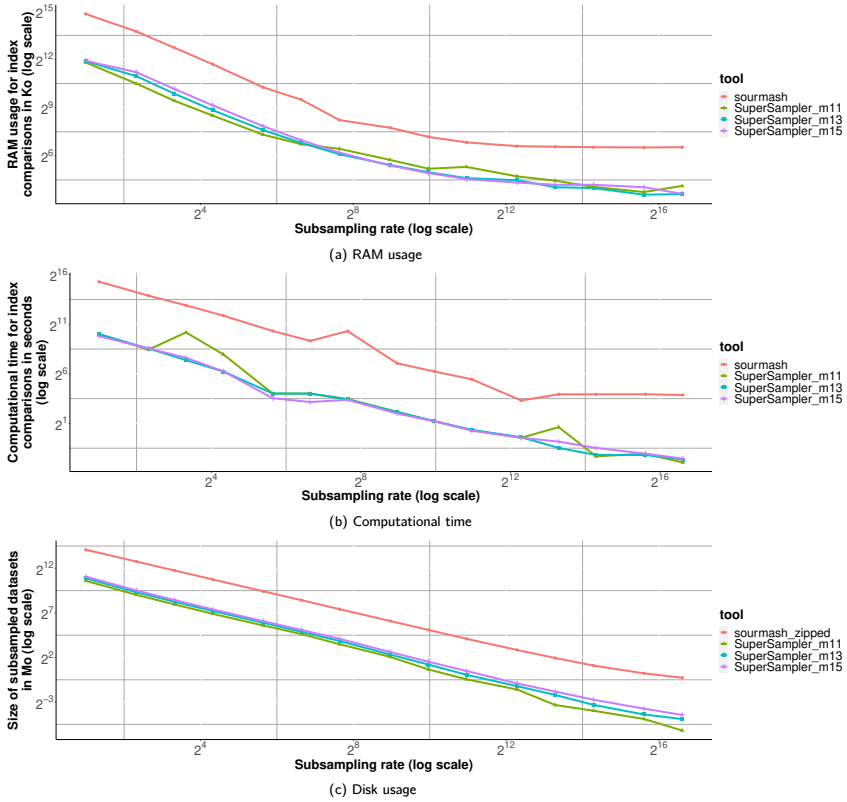


Figure 17.8.: Resource consumption results for 1024 bacterial genomes from RefSeq with $k = 31$.

17.4.2.1. RAM, time, and sketch size

Figure 17.8 and Figure E.3 demonstrate that, for $k = 31$, SuperSampler generally consumes 5 times less RAM and requires 16 times less space than sourmash. Additionally, SuperSampler performs computations 50 times faster than sourmash when comparing highly dissimilar genomes. However, when genomes are very similar, such as with Salmonella, comparison times are comparable since SuperSampler's time optimization does not apply to very similar documents. We also note that minimizer size has little to no impact on these metrics.

Figure E.4 and Figure E.5 reveal that the improvement in sketch disk size is even more significant with larger values of k. SuperSampler uses in general 50 times less disk space than sourmash with $k = 63$, while maintaining similar differences in RAM and computation time.

17.4.2.2. Error

SuperSampler's accuracy is slightly below sourmash's (Figure 17.9), due to a mild clustering effect: k-mers sharing a small minimizer tend to be co-selected. In terms of precision per unit of compressed sketch size, however, SuperSampler compares favorably (Figure 17.10). SuperSampler also stores k-mers exactly, so shared k-mers can be retrieved directly, something sourmash cannot do without switching to larger, invertible hashes.

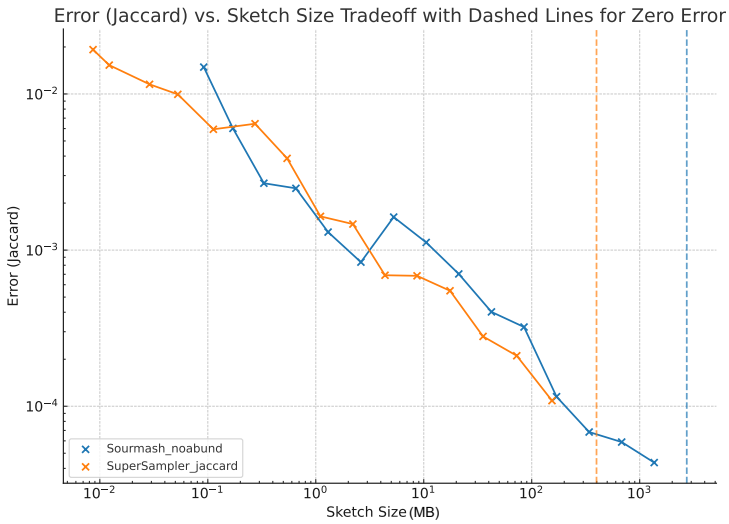


Figure 17.9.: Error Jaccard similarity approximation for sourmash and SuperSampler against the compressed sketch sizes. This plot shows the results for 20 files of salmonellas simulated reads with $k = 31$, $m = 13$. Reads are 150bp long for a 100X coverage. Dashed lines represent the sizes of the sketches indexing all k-mers.

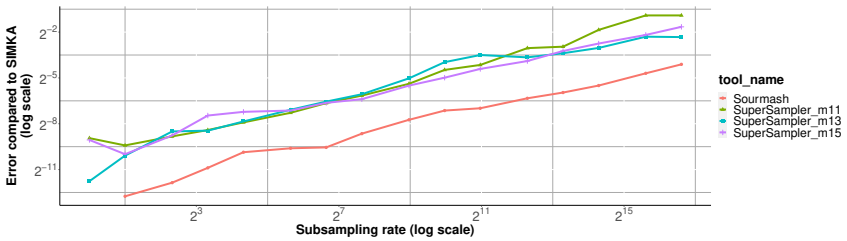


Figure 17.10.: Error against Simka on Containment similarity approximation for sourmash (red line) and SuperSampler with different minimizer sizes, on 1024 Salmonella genomes with $k = 63$.

17.4.2.3. Massive collection indexing

Figure 17.11 shows performance on 100 to 128,000 RefSeq bacterial genomes. SuperSampler handles the full range in under 25 CPU hours; sourmash was stopped at 32,000 genomes. The performance gap narrows at large scale because the $n \times n$ output matrix dominates memory pressure for both tools. Sketch creation is cheap in both cases, with IO as the bottleneck.

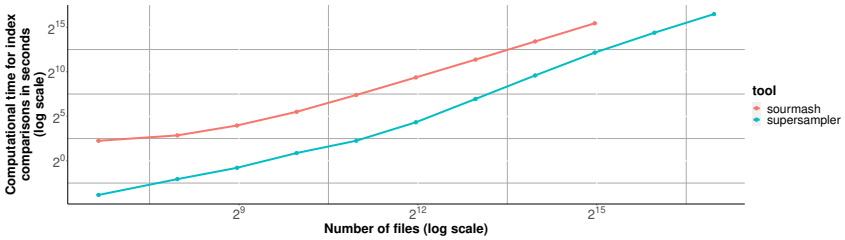


Figure 17.11.: Computational time for comparisons on different amounts of bacterial genomes from RefSeq. From 100 to 128,000 genomes with $k = 63$, $s = 1000$ and $m = 15$. sourmash was run up to 32,000 genomes as it was taking too much time for the 2 last experiments.

17.5. Conclusion

This chapter introduced *fractional hitting sets*, a relaxation of universal hitting sets that covers a fraction f of the k-mer space. The resulting small-minimizer scheme achieves a density factor below 1.5 for $f \leq 0.8$ and approaching 1 as $f \to 0$, producing longer and more compact super-k-mers than full-coverage schemes, with an increasing proportion of maximal super-k-mers as f decreases. SuperSampler exploits this by storing selected super-k-mers in minimizer-partitioned sketches, enabling both compact encoding and efficient partition-level comparisons. Experiments confirm the theoretical density bounds and show substantially lower disk and RAM usage than sourmash at comparable accuracy.

We plan to investigate alternative methods for sketch comparison, like sorted fingerprints, which could potentially reduce the complexity of the comparison process. From a theoretical perspective, delving deeper into the properties of fractional hitting sets and gaining a better understanding of density and restricted density bounds for various values of f may lead to even more efficient and robust sketching techniques. Finally, studying theoretical guarantees on the bias of this approach would be an interesting direction.

Third discussion

The starting point of this last part is that, at the scale of data we now have to deal with, we cannot afford to keep every k-mer. Estimating similarity does not require exhaustive indexing, and the previous parts already pushed exact representations close to their limit, both in terms of throughput and space. Sparsifying the input is therefore a natural next step, and one that becomes more and more important as collections grow. The two chapters of this part covered complementary angles on this question. Chapter 16 tackled it from the minimizer side, asking how far we can push the density of a window-based scheme by allowing several candidates per window. Chapter 17 approached it from the sketching side, introducing fractional hitting sets to relax universal coverage while exploiting the super-k-mer structure of the selected k-mers. Both share the same broader goal of selecting fewer k-mers while preserving the information that matters for downstream comparisons.

A first observation I want to start with is that recent work on minimizers, my own included, has been very focused on density. There is a good reason for this, since lower density translates almost directly into smaller indexes, but density is not the only thing that matters. Metrics like conservation, the fraction of selected k-mers that survive mutations, are just as important when it comes to comparing the similarity of sequences. If density captures the "more with less" side of sampling, conservation captures the "well" side.

Another angle I find under-explored is the spectrum between context-free schemes such as FracMinHash and context-based ones such as minimizers or syncmers. Most work sits at one of the two ends, but there is no reason they cannot be combined. Chapter 17 was a first step in that direction, layering a hash-based subsampling on top of minimizer selection, in the same spirit as MashMap [Jai+17] which combines MinHash sketches with minimizers for long-read mapping. I am convinced this middleground is promising and has more to offer than density alone.

A direction I am currently exploring along similar lines is what I tentatively call lexicographic-informed sketching. All the sketching methods presented in Chapter 3 treat k-mers as opaque hashes, completely losing their string structure. Drawing inspiration from LexicHash [GRS23] and LexicMap [SLI25], we are exploring sketches that rely on the actual k-mer strings to drive the sampling, replacing the binary "match or not" comparison of hashes with a more granular notion of string match. My intuition is that this should improve the precision of sketches compared to MinHash and unlock new optimization opportunities for sketch storage and comparison.

A related line of recent work, briefly discussed in Chapter 3 and Chapter 4, is the design of data structures in *sketch space*, working directly on the sequence of selected k-mers rather than on the original sequence. Minimizer-space de Bruijn graphs [EBC21; Eki+23] and the U-index [Aya+25] are recent examples that illustrate how much can be saved by indexing shorter sequences over a larger alphabet. This is the angle of another ongoing collaboration with Rob Patro on scalable indexing through locally consistent phrases. The central idea is to index the content *between* the selected anchors rather than the anchors themselves. Here the choice of anchors directly shapes the distribution of distances between them. Context-free criteria such as prefix-free parsing give a roughly geometric distribution with no upper bound, while minimizers let us pick the upper bound through the window size but offer no lower bound beyond 1. The middleground we are looking for would combine moderate context dependence with a reasonably balanced distribution, and an interesting problem is whether we can enforce both a lower and an upper bound on the distance between anchors.

One question that came out of this phrase-indexing approach, and that I find genuinely

interesting, is whether we can build good compact sketches to compare small sets, on the order of a hundred k-mers per phrase. This regime does not seem to have been studied much, as most sketching work has focused on whole genomes or large metagenomes, but small-set sketches would be ideal to compare individual phrases at low cost. I expect that the right answer will not be a scaled-down MinHash but something closer in spirit to DotHash [Nun+23], where the comparison naturally produces a numerical similarity rather than an exact set operation.

Looking across all these directions, my overarching belief is that we should explore more algorithms and data structures in sketch space.

Conclusion & perspectives

Understanding living organisms at the molecular level has always required reading their genomes, and reading genomes has meant more and more dealing with massive amounts of sequence data. Over the past two decades, high-throughput sequencing has turned into a computational problem where public archives now hold more than fifty petabases of raw reads, and the bottleneck has long since shifted from producing data to analyzing it.

At the heart of most analysis pipelines sits a simple object, the k-mer, a substring of fixed length k. Counting these k-mers, indexing them, comparing sets of them have been operations that underlie genome assembly, read mapping, metagenomics, and large-scale similarity search. After two decades of research and thousands of tools built around them, what remained to be done was a legitimate question, and one of the first I got asked when presenting my work at the start of my PhD was:

> Why do we keep studying k-mers nowadays?

I remember being quite confused at the time, unsure how to defend a research object that had been around for so long. Three years later, I think this thesis offers part of an answer.

The three parts of the manuscript each take a different angle on the same broader problem. The first asked how far we could push the throughput of a genomic pipeline when the algorithm and the implementation are designed together, and showed that careful vectorization across parsing, hashing and minimizer selection brought us close to the hardware limit. The second asked how the representation of a k-mer set could itself become a source of performance, and showed that structures exploiting the locality between consecutive k-mers improve both space and streaming access while widening the range of operations a dictionary can natively support. The third asked how much we could afford to drop from the input, and how to choose what to keep so that similarity queries still hold on much sparser representations.

The three parts share the same underlying argument, that sequence and representation should be treated as a single design problem rather than two separate ones, and this holds whether we look at the raw stream of nucleotides, at k-mer-based dictionaries, or at the sparser sketches that summarize them. The third part pushes this idea one step further by suggesting that the sampled view of a sequence deserves its own data structures, its own indexes and its own primitives, rather than being seen as a lossy preprocessing step before going back to k-mer-level analysis.

Stepping back from the technical content, our field has evolved a lot between the start of my PhD and today. So has academia at large, and computer science more broadly. The most visible shift is of course the rapid adoption of large language models. They have changed the everyday work of software developers in ways that would have felt surprising three years ago, and are now part of the toolbox of many. I do not believe transformers are the answer to every single problem, and they remain quite challenging to apply at the scale of sequencing data we work with. They have however driven a broader interest in embeddings and vector representations, and I think this opens avenues worth exploring for our community.

There is for instance a loose connection between the way text embeddings work and the way we compute sequence fingerprints. Both turn a long stream into a compact representation that captures something about its content, and both lend themselves to nearest-neighbor search. Whether vector search techniques can be adapted to minimizer or sketch embeddings is, to me,

an interesting question, and I would not be surprised if useful crossover ideas appeared in the coming years.

These changes also reflect on the way we teach and learn computer science. As a teaching assistant during my PhD, I had to rethink how to evaluate students, and how to help them build the intuition they need before reaching for a language model on every problem. The same applies to software development more broadly. Lowering the barrier to entry has made many projects easier to start, but it has also made codebases harder to keep maintainable. I worry that the open-source ecosystem may end up flooded by short-lived libraries that nobody really maintains, and I see a real need to foster a few well-crafted community tools instead of reinventing them under slightly different names.

What I find most interesting in the shift of the last few years is actually not the software side, but the hardware side. Training and running these models at the scale of trillions of parameters has pushed the design of new accelerators, memory hierarchies and interconnects much further than what we would have seen otherwise. A good example is the unified memory model now found on Apple Silicon and on several recent server-grade machines, where the CPU and GPU share the same physical memory and can exchange data with virtually no cost. This design was largely motivated by the needs of model inference, but the underlying improvements are a real opportunity for sequence processing too. If these hardware advances reach consumer machines, they could finally make GPU acceleration practical for the kind of pipelines I worked on during this thesis.

Paradoxically, I think this same trend makes frugal algorithmic solutions more important than ever, not less. Energy and compute are scarce wherever the demand keeps growing, and most of the new sequencing data will not be analyzed in a centralized cluster but on whatever workstation happens to be available. The compromise I see emerging is a mix of frugal local computing on lightweight summaries and a few very optimized centralized indexes that everyone can query. The Logan project, which compacts most of the Sequence Read Archive into a searchable index of unitigs, is a nice early example of what the second half can look like. Pushing on the first half is where I see the most room left, and I am convinced the future is sparse, with more and more computation happening directly on sketches rather than on the full k-mer content. Mapping and assembly already have early proofs of concept in that direction, and I think there is a lot more to push in that direction.

The last observation I want to end on is that this thesis stayed firmly on the DNA side of the problem. Tremendous progress has happened on protein analysis in parallel, with deep learning reshaping what we can extract from amino acid sequences, and recent work on hybrid methods combining the two has been quite promising. I find this convergence very exciting. Sparse, anchor-based structures for large collections are starting to appear across both communities, whether we look at de Bruijn graphs of minimizers, syncmers or amino acids, and I think there is a lot to learn from putting these two worlds in the same conversation. This is a direction I would like to explore after this PhD.

References

[ABKU94] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. “Balanced allocations (extended abstract)”. In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing - STOC ’94*. STOC ’94. ACM Press, 1994, pp. 593–602. DOI: 10.1145/195058.195412. URL: http://dx.doi.org/10.1145/195058.195412 → 133.

[ABP23] Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. “Longest Common Prefix Arrays for Succinct k-Spectra”. In: *String Processing and Information Retrieval*. Springer Nature Switzerland, 2023, pp. 1–13. ISBN: 9783031439803. DOI: 10.1007/978-3-031-43980-3_1. URL: http://dx.doi.org/10.1007/978-3-031-43980-3_1 → 87.

[ABP25] Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. “Finimizers: Variable-Length Bounded-Frequency Minimizers for k-mer Sets”. In: *IEEE Transactions on Computational Biology and Bioinformatics* 22.2 (Mar. 2025), pp. 899–910. ISSN: 2998-4165. DOI: 10.1109/tcbbio.2025.3545285. URL: http://dx.doi.org/10.1109/TCBBIO.2025.3545285 → 78, 130.

[ADMP26] Jarno N. Alanko, Lore Depuydt, Camille Marchet, and Simon J. Puglisi. “Fast Set Operations for Compact k-mer Sets”. In: *bioRxiv* (May 2026). DOI: 10.64898/2026.05.24.727514. URL: http://dx.doi.org/10.64898/2026.05.24.727514 → 123.

[Ala+21] Jarno Alanko, Bahar Alipanahi, Jonathen Settle, Christina Boucher, and Travis Gagie. “Buffering updates enables efficient dynamic de Bruijn graphs”. In: *Computational and Structural Biotechnology Journal* 19 (2021), pp. 4067–4078. ISSN: 2001-0370. DOI: 10.1016/j.csbj.2021.06.047. URL: http://dx.doi.org/10.1016/j.csbj.2021.06.047 → 94.

[AM24] Md. Hasin Abrar and Paul Medvedev. “PLA-index: A k-mer Index Exploiting Rank Curve Linearity”. en. In: vol. 312. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 13:1–13:18. DOI: 10.4230/LIPICS.WABI.2024.13. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.13 → 96.

[APV23] Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. “Small Searchable 𝑘-Spectra via Subset Rank Queries on the Spectral Burrows-Wheeler Transform”. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. Society for Industrial and Applied Mathematics, Jan. 2023, pp. 225–236. ISBN: 9781611977714. DOI: 10.1137/1.9781611977714.20. URL: http://dx.doi.org/10.1137/1.9781611977714.20 → 69.

[Arm26] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. https://developer.arm.com/documentation/ddi0487/latest/. Arm Limited. 2026. URL: https://developer.arm.com/documentation/ddi0487/latest/ → 34.

[Atk08] Kendall E Atkinson. *An introduction to numerical analysis, 2nd ed.* en. New York, NY: John Wiley & Sons, Sept. 2008 → 197.

[Aya+25] Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P. Pissis. “U-Index: A Universal Indexing Framework for Matching Long Patterns”. en. In: vol. 338. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, July 2025, 4:1–4:18. DOI: 10.4230/LIPICS.SEA.2025.4. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2025.4 → 29, 157.

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. “Adding Virtualization Capabilities to the Grid’5000 Testbed”. In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1 → 44.

[Ban+12] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. “SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing”. In: *Journal of Computational Biology* 19.5 (Apr. 2012), pp. 455–477. ISSN: 1557-8666. DOI: 10.1089/cmb.2012.0021. URL: http://dx.doi.org/10.1089/cmb.2012.0021 → 16.

[Ban+22] Anton Bankevich, Andrey V. Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A. Pevzner. “Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads”. In: *Nature Biotechnology* 40.7 (Feb. 2022), pp. 1075–1081. ISSN: 1546-1696. DOI: 10.1038/s41587-022-01220-6. URL: http://dx.doi.org/10.1038/s41587-022-01220-6 → 111.

[BBK21] Karel Břinda, Michael Baym, and Gregory Kucherov. “Simplitigs as an efficient and scalable representation of de Bruijn graphs”. In: *Genome Biology* 22.1 (Apr. 2021). ISSN: 1474-760X. DOI: 10.1186/s13059-021-02297-z. URL: http://dx.doi.org/10.1186/s13059-021-02297-z → 16.

[BCEG17] Philip Bille, Anders Roy Christiansen, Mikko Berggren Ettienne, and Inge Li Gørtz. “Fast Dynamic Arrays”. en. In: vol. 87. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 16:1–16:13. DOI: 10.4230/LIPICS.ESA.2017.16. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2017.16 → 93.

[Ben+16] Gaëtan Benoit, Pierre Peterlongo, Mahendra Mariadassou, Erwan Drezen, Sophie Schbath, Dominique Lavenier, and Claire Lemaitre. “Multiple comparative metagenomics using multiset k-mer counting”. In: *PeerJ Computer Science* 2 (Nov. 2016), e94. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.94. URL: http://dx.doi.org/10.7717/peerj-cs.94 → 153.

[Ben+24] Gaëtan Benoit, Sébastien Raguideau, Robert James, Adam M. Phillippy, Rayan Chikhi, and Christopher Quince. “High-quality metagenome assembly from long accurate reads with metaMDBG”. In: *Nature Biotechnology* 42.9 (Jan. 2024), pp. 1378–1383. ISSN: 1546-1696. DOI: 10.1038/s41587-023-01983-6. URL: http://dx.doi.org/10.1038/s41587-023-01983-6 → 116.

[BL19] Daniel N. Baker and Ben Langmead. “Dashing: fast and accurate genomic distances with HyperLogLog”. In: *Genome Biology* 20.1 (Dec. 2019). ISSN: 1474-760X. DOI: 10.1186/s13059-019-1875-0. URL: http://dx.doi.org/10.1186/s13059-019-1875-0 → 22.

[BL23] Daniel N Baker and Ben Langmead. “Genomic sketching with multiplicities and locality-sensitive hashing using Dashing 2”. In: Cold Spring Harbor Laboratory, July 2023, gr.277655.123. DOI: 10.1101/gr.277655.123. URL: http://dx.doi.org/10.1101/gr.277655.123 → 22.

[BMAP24] Anthony Baire, Pierre Marijon, Francesco Andreace, and Pierre Peterlongo. “Back to sequences: Find the origin of k-mers”. In: *Journal of Open Source Software* 9.101 (2024), p. 7066. DOI: 10.21105/joss.07066. URL: https://doi.org/10.21105/joss.07066 → 69, 70, 72.

[BOSS12] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. “Succinct de Bruijn Graphs”. In: *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2012, pp. 225–235. ISBN: 9783642331220. DOI: 10.1007/978-3-642-33122-0_18. URL: http://dx.doi.org/10.1007/978-3-642-33122-0_18 → 86.

[Bra+19] Phelim Bradley, Henk C. den Bakker, Eduardo P. C. Rocha, Gil McVean, and Zamin Iqbal. “Ultrafast search of all deposited bacterial and viral genomic data”. In: *Nature Biotechnology* 37.2 (Feb. 2019), pp. 152–159. ISSN: 1546-1696. DOI: 10.1038/s41587-018-0010-1. URL: http://dx.doi.org/10.1038/s41587-018-0010-1 → 17.

[Bro97] A.Z. Broder. “On the resemblance and containment of documents”. In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. SEQUEN-97. IEEE Comput. Soc, June 1997, pp. 21–29. DOI: 10.1109/sequen.1997.666900. URL: http://dx.doi.org/10.1109/SEQUEN.1997.666900 → 20, 21.

[BS10] Ricardo Baeza-Yates and Alejandro Salinger. “Fast Intersection Algorithms for Sorted Sequences”. In: *Algorithms and Applications*. Springer Berlin Heidelberg, 2010, pp. 45–61. ISBN: 9783642124761. DOI: 10.1007/978-3-642-12476-1_3. URL: http://dx.doi.org/10.1007/978-3-642-12476-1_3 → 102.

[BW94] Michael Burrows and David J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Technical Report 124. Digital Equipment Corporation, 1994 → 15.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. “An almost optimal algorithm for unbounded searching”. In: *Information Processing Letters* 5.3 (Aug. 1976), pp. 82–87. ISSN: 0020-0190. DOI: 10.1016/0020-0190(76)90071-5. URL: http://dx.doi.org/10.1016/0020-0190(76)90071-5 → 102.

[Che+26] Ke Chen, Xiang Li, Qian Shi, Mingfu Shao, and Paul Medvedev. “Hash functions in nucleotide sequence analysis”. In: *Genome Research* (May 2026). ISSN: 1549-5469. DOI: 10.1101/gr.281453.125. URL: http://dx.doi.org/10.1101/gr.281453.125 → 22.

[Chi+15] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. “On the Representation of De Bruijn Graphs”. In: *Journal of Computational Biology* 22.5 (Jan. 2015), pp. 336–352. ISSN: 1557-8666. DOI: 10.1089/cmb.2014.0160. URL: http://dx.doi.org/10.1089/cmb.2014.0160 → 17, 28.

[Chi+24] Rayan Chikhi, Téo Lemane, Raphaël Loll-Krippleber, Mercè Montoliu-Nerin, Brice Raffestin, Antonio Pedro Camargo, Carson J. Miller, Mateus Bernabe Fiamenghi, Daniel Paiva Agustinho, Sina Majidian, Greg Autric, Maxime Hugues, Junkyoung Lee, Roland Faure, Kristen D. Curry, Jorge A. Moura de Sousa, Eduardo P. C. Rocha, David Koslicki, Paul Medvedev, Purav Gupta, Jessica Shen, Alejandro Morales-Tapia, Kate Sihuta, Peter J. Roy, Grant W. Brown, Robert C. Edgar, Anton Korobeynikov, Martin Steinegger, Caleb A. Lareau, Pierre Peterlongo, and Artem Babaian. “Logan: Planetary-Scale Genome Assembly Surveys Life’s Diversity”. In: *bioRxiv* (July 2024). DOI: 10.1101/2024.07.30.605881. URL: http://dx.doi.org/10.1101/2024.07.30.605881 → 18, 76.

[CLC25] Bede Constantinides, John Lees, and Derrick W Crook. “Deacon: fast sequence filtering and contaminant depletion”. In: *bioRxiv* (2025). DOI: 10.1101/2025.06.09.658732 → 69.

[CLM16] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. “Compacting de Bruijn graphs from sequencing data quickly and in low memory”. In: *Bioinformatics* 32.12 (June 2016), pp. i201–i208. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw279. URL: http://dx.doi.org/10.1093/bioinformatics/btw279 → 17, 127.

[Coh97] Jonathan D. Cohen. “Recursive hashing functions for n-grams”. In: *ACM Transactions on Information Systems* 15.3 (July 1997), pp. 291–320. ISSN: 1558-2868. DOI: 10.1145/256163.256168. URL: http://dx.doi.org/10.1145/256163.256168 → 50.

[CPFP24] Alessio Campanelli, Giulio Ermanno Pibiri, Jason Fan, and Rob Patro. “Where the Patterns Are: Repetition-Aware Compression for Colored de Bruijn Graphs”. In: *Journal of Computational Biology* 31.10 (Oct. 2024), pp. 1022–1044. ISSN: 1557-8666. DOI: 10.1089/cmb.2024.0714. URL: http://dx.doi.org/10.1089/cmb.2024.0714 → 18.

[CPS25] Ke Chen, Vinamratha Pattar, and Mingfu Shao. “Sequence Similarity Estimation by Random Subsequence Sketching”. en. In: vol. 344. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Aug. 2025, 7:1–7:17. DOI: 10.4230/LIPICS.WABI.2025.7. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2025.7 → 23.

[Cro+99] Maxime Crochemore, A. Czumaj, L. Ga▯sieniec, T. Lecroq, W. Plandowski, and W. Rytter. “Fast practical multi-pattern matching”. In: *Information Processing Letters* 71.3 (1999), pp. 107–113. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(99)00092-7 → 69.

[Cro25] Nicholas D. Crosbie. “grepq: A Rust application that quickly filters FASTQ files by matching sequences to a set of regular expressions”. In: *Journal of Open Source Software* 10.110 (June 2025), p. 8048. ISSN: 2475-9066. DOI: 10.21105/joss.08048. URL: http://dx.doi.org/10.21105/joss.08048 → 69, 72.

[CT23] Andrea Cracco and Alexandru I. Tomescu. “Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT”. In: *Genome Research* (May 2023). ISSN: 1549-5469. DOI: 10.1101/gr.277615.122. URL: http://dx.doi.org/10.1101/gr.277615.122 → 17, 28, 127.

[Dar+22] Mitra Darvish, Enrico Seiler, Svenja Mehringer, René Rahn, and Knut Reinert. “Needle: a fast and space-efficient prefilter for estimating the quantification of very large collections of expression experiments”. In: *Bioinformatics* 38.17 (July 2022). Ed. by Yann Ponty, pp. 4100–4108. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btac492. URL: http://dx.doi.org/10.1093/bioinformatics/btac492 → 28, 70, 137.

[DDG13] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. “Disk-based k-mer counting on a PC”. In: *BMC Bioinformatics* 14.1 (May 2013). ISSN: 1471-2105. DOI: 10.1186/1471-2105-14-160. URL: http://dx.doi.org/10.1186/1471-2105-14-160 → 115.

[DGD24] Yoann Dufresne, Vincent Guillemot, and Johann Dreo. “Optimization of Reversible Hash Functions for k-mer data structures”. In: *SeqBIM 2024 Workshop*. Rennes, France, Nov. 2024. URL: http://seqbim.cnrs.fr/wp-content/uploads/2024/11/Proceedings_SeqBIM_2024.pdf → 79.

[DKGD15] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. “KMC 2: fast and resource-frugal k-mer counting”. In: *Bioinformatics* 31.10 (Jan. 2015), pp. 1569–1576. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btv022. eprint: https://academic.oup.com/bioinformatics/article-pdf/31/10/1569/49012901/bioinformatics_31_10_1569.pdf. URL: https://doi.org/10.1093/bioinformatics/btv022 → 28, 115.

[DLM00] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. “Adaptive set intersections, unions, and differences”. In: *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’00. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 2000, pp. 743–752. ISBN: 0898714532. URL: https://dl.acm.org/doi/10.5555/338219.338634 → 102.

[DLS24] Diego Díaz-Domínguez, Miika Leinonen, and Leena Salmela. “Space-efficient computation of k-mer dictionaries for large values of k”. In: *Algorithms Mol. Biol.* 19.1 (2024), p. 14. DOI: 10.1186/S13015-024-00259-1. URL: https://doi.org/10.1186/s13015-024-00259-1 → 115.

[DPM25] Bastien Degardins, Charles Paperman, and Camille Marchet. “Vizitig: a pangenome and pantranscriptome explorer”. In: *bioRxiv* (Apr. 2025). DOI: 10.1101/2025.04.19.649656. URL: http://dx.doi.org/10.1101/2025.04.19.649656 → 17, 123.

[DPR22] Saska Donges, Simon J. Puglisi, and Rajeev Raman. “On Dynamic Bitvector Implementations”. In: *2022 Data Compression Conference (DCC)*. IEEE, Mar. 2022, pp. 252–261. DOI: 10.1109/dcc52660.2022.00033. URL: http://dx.doi.org/10.1109/DCC52660.2022.00033 → 92.

[EBC21] Barış Ekim, Bonnie Berger, and Rayan Chikhi. “Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer”. In: *Cell Systems* 12.10 (Oct. 2021), 958–968.e6. ISSN: 2405-4712. DOI: 10.1016/j.cels.2021.08.009. URL: http://dx.doi.org/10.1016/j.cels.2021.08.009 → 16.

[Edg21] Robert Edgar. “Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences”. In: *PeerJ* 9 (Feb. 2021), e10805. ISSN: 2167-8359. DOI: 10.7717/peerj.10805. URL: http://dx.doi.org/10.7717/peerj.10805 → 21, 27, 111.

[Eki+23] Bariş Ekim, Kristoffer Sahlin, Paul Medvedev, Bonnie Berger, and Rayan Chikhi. “Efficient mapping of accurate long reads in minimizer space with mapquik”. In: *Genome Research* (June 2023). ISSN: 1549-5469. DOI: 10.1101/gr.277679.123. URL: http://dx.doi.org/10.1101/gr.277679.123 → 29, 70, 157.

[Eli74] Peter Elias. “Efficient Storage and Retrieval by Content and Address of Static Files”. In: *Journal of the ACM* 21.2 (Apr. 1974), pp. 246–260. ISSN: 1557-735X. DOI: 10.1145/321812.321820. URL: http://dx.doi.org/10.1145/321812.321820 → 84.

[ERM17] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. “Gerbil: a fast and memory-efficient k-mer counter with GPU-support”. In: *Algorithms for Molecular Biology* 12.1 (Mar. 2017). ISSN: 1748-7188. DOI: 10.1186/s13015-017-0097-9. URL: http://dx.doi.org/10.1186/s13015-017-0097-9 → 115.

[Fan+24] Jason Fan, Jamshed Khan, Noor Pratap Singh, Giulio Ermanno Pibiri, and Rob Patro. “Fulgor: a fast and compact k-mer index for large-scale matching and color queries”. In: *Algorithms for Molecular Biology* 19.1 (Jan. 2024), p. 3. ISSN: 1748-7188. DOI: 10.1186/s13015-024-00251-9. URL: https://doi.org/10.1186/s13015-024-00251-9 (visited on 10/18/2024) → 18.

[Fan71] Robert Mario Fano. *On the Number of Bits Required to Implement an Associative Memory*. Tech. rep. Memorandum 61. Cambridge, MA: Computer Structures Group, MIT, 1971. URL: http://csg.csail.mit.edu/pubs/memos/Memo-61/Memo-61.pdf → 84.

[Fau+25] Roland Faure, Hasin Abrar, Haonan Wu, Rayan Chikhi, David Koslicki, and Paul Medvedev. "Comparing and indexing metagenomes at a large scale using random projections". In: *SeqBIM 2025 Workshop*. Nantes, France, Nov. 2025. URL: https://hal.science/hal-05412009 → 23.

[FCPL22] Xiaowen Feng, Haoyu Cheng, Daniel Portik, and Heng Li. "Metagenome assembly of high-fidelity long reads with hifiasm-meta". In: *Nature Methods* 19.6 (May 2022), pp. 671–674. ISSN: 1548-7105. DOI: 10.1038/s41592-022-01478-3. URL: http://dx.doi.org/10.1038/s41592-022-01478-3 → 116.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH,...Proceedings (Jan. 2007). ISSN: 1365-8050. DOI: 10.46298/dmtcs.3545. URL: http://dx.doi.org/10.46298/dmtcs.3545 → 22.

[FL13] Simone Faro and Thierry Lecroq. "The exact online string matching problem: A review of the most recent results". In: *ACM Computing Surveys* 45.2 (Feb. 2013), pp. 1–42. ISSN: 1557-7341. DOI: 10.1145/2431211.2431212. URL: http://dx.doi.org/10.1145/2431211.2431212 → 69.

[FM00] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. SFCS-00. IEEE Comput. Soc, Nov. 2000, pp. 390–398. DOI: 10.1109/sfcs.2000.892127. URL: http://dx.doi.org/10.1109/SFCS.2000.892127 → 15.

[Gal24] Andrew Gallant. *ripgrep*. https://github.com/BurntSushi/ripgrep. 2024. URL: https://github.com/BurntSushi/ripgrep → 69, 72.

[GI24] Ragnar Groot Koerkamp and Pesho Ivanov. "Exact global alignment using A* with chaining seed heuristic and match pruning". In: *Bioinformatics* 40.3 (Jan. 2024). Ed. by Tobias Marschall. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae032. URL: http://dx.doi.org/10.1093/bioinformatics/btae032 → 14.

[GLP25] Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri. "The open-closed mod-minimizer algorithm". In: *Algorithms for Molecular Biology* 20.1 (Mar. 2025). ISSN: 1748-7188. DOI: 10.1186/s13015-025-00270-0. URL: http://dx.doi.org/10.1186/s13015-025-00270-0 → 78.

[GM25] Ragnar Groot Koerkamp and Igor Martayan. "SimdMinimizers: Computing Random Minimizers, fast". In: *23rd International Symposium on Experimental Algorithms (SEA 2025)*. Vol. 338. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, July 2025. DOI: 10.4230/LIPIcs.SEA.2025.20 → 47.

[GMP23] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. "Supporting Descendants in SIMD-Accelerated JSONPath". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS '23. ACM, Mar. 2023, pp. 338–361. DOI: 10.1145/3623278.3624754. URL: http://dx.doi.org/10.1145/3623278.3624754 → 40.

[GMS17] Travis Gagie, Giovanni Manzini, and Jouni Sirén. "Wheeler graphs: A framework for BWT-based data structures". In: *Theoretical Computer Science* 698 (Oct. 2017), pp. 67–78. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2017.06.016. URL: http://dx.doi.org/10.1016/j.tcs.2017.06.016 → 86.

[Gol+25] Shay Golan, Ido Tziony, Matan Kraus, Yaron Orenstein, and Arseny Shur. "GreedyMini: generating low-density DNA minimizers". In: *Bioinformatics* 41.Supplement_1 (July 2025), pp. i275–i284. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaf251. URL: http://dx.doi.org/10.1093/bioinformatics/btaf251 → 78, 130.

[GP13] Simon Gog and Matthias Petri. "Optimized succinct data structures for massive data". In: *Software: Practice and Experience* 44.11 (May 2013), pp. 1287–1314. ISSN: 1097-024X. DOI: 10.1002/spe.2198. URL: http://dx.doi.org/10.1002/spe.2198 → 92.

[GP24] Ragnar Groot Koerkamp and Giulio Ermanno Pibiri. "The mod-minimizer: A Simple and Efficient Sampling Algorithm for Long k-mers". In: *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*. Ed. by Solon P. Pissis and Wing-Kin Sung. Vol. 312. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 11:1–11:23. ISBN: 978-3-95977-340-9. DOI: 10.4230/LIPIcs.WABI.2024.11. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.11 → 64.

[Gra93] Goetz Graefe. "Query evaluation techniques for large databases". In: *ACM Computing Surveys* 25.2 (June 1993), pp. 73–169. ISSN: 1557-7341. DOI: 10.1145/152610.152611. URL: http://dx.doi.org/10.1145/152610.152611 → 102.

[Gro24] Ragnar Groot Koerkamp. "A*PA2: Up to 19× Faster Exact Global Alignment". en. In: vol. 312. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 17:1–17:25. DOI: 10.4230/LIPICS.WABI.2024.17. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.17 → 14.

[Gro25a] Ragnar Groot Koerkamp. "Optimal Throughput Bioinformatics". en. PhD thesis. 2025. DOI: 10.3929/ETHZ-C-000783091. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/783091 → 14.

[Gro25b] Ragnar Groot Koerkamp. "PtrHash: Minimal Perfect Hashing at RAM Throughput". en. In: vol. 338. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 21:1–21:21. DOI: 10.4230/LIPICS.SEA.2025.21. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2025.21 → 84.

[Gro26] Ragnar Groot Koerkamp. "The anti-lexicographic SUS-anchor: a near-optimal k=1 sampling scheme". In: *arXiv* (2026). DOI: 10.48550/arXiv.2606.01190. arXiv: 2606.01190 [cs.DS]. URL: https://arxiv.org/abs/2606.01190 → 130.

[GRS23] Grant Greenberg, Aditya Narayan Ravi, and Ilan Shomorony. "LexicHash: sequence similarity estimation via lexicographic comparison of hashes". In: *Bioinformatics* 39.11 (Oct. 2023). Ed. by Can Alkan. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btad652. URL: http://dx.doi.org/10.1093/bioinformatics/btad652 → 157.

[Her+25] Yohan Hernandez-Courbevoie, Mikaël Salson, Chloé Bessière, Haoliang Xue, Daniel Gautheret, Camille Marchet, and Antoine Limasset. "REINDEER2: Practical Abundance Index at Scale". In: *String Processing and Information Retrieval*. Springer Nature Switzerland, Sept. 2025, pp. 156–171. ISBN: 9783032052285. DOI: 10.1007/978-3-032-05228-5_14. URL: http://dx.doi.org/10.1007/978-3-032-05228-5_14 → 17.

[HKM25] Mahmudur Rahman Hera, David Koslicki, and Conrado Martínez. "MaxGeomHash: An Algorithm for Variable-Size Random Sampling of Distinct Elements". In: *bioRxiv* (Nov. 2025). DOI: 10.1101/2025.11.11.687920. URL: http://dx.doi.org/10.1101/2025.11.11.687920 → 22.

[HM20] Guillaume Holley and Páll Melsted. “Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs”. In: *Genome Biology* 21.1 (Sept. 2020). ISSN: 1474-760X. DOI: 10.1186/s13059-020-02135-8. URL: http://dx.doi.org/10.1186/s13059-020-02135-8 → 28, 58, 94.

[Hom+25] Nils Homer, Seth Stadick, Shea Lambert, Matt Stone, and Tim Fennell. *fqgrep*. https://github.com/fulcrumgenomics/fqgrep. Version 1.1.1. Mar. 2025. DOI: 10.5281/zenodo.15034074. URL: https://doi.org/10.5281/zenodo.15034074 → 69, 72.

[HPK26] John L. Hennessy, David A. Patterson, and Christos Kozyrakis. *Computer Architecture: A Quantitative Approach*. Seventh edition. Cambridge, MA: Morgan Kaufmann Publishers, 2026. ISBN: 978-0-443-15406-5. URL: https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-443-15406-5 → 33.

[HST17] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. “Sliding-Window Aggregation Algorithms: Tutorial”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*. ACM, 2017, pp. 11–14. DOI: 10.1145/3093742.3095107. URL: https://doi.org/10.1145/3093742.3095107 → 59.

[ILMS26] Florian Ingels, Antoine Limasset, Camille Marchet, and Mikaël Salson. “Vigemers: on the number of k-mers sharing the same XOR-based minimizer”. In: *arXiv* (2026). DOI: 10.48550/arXiv.2602.03337. arXiv: 2602.03337 [cs.DM]. URL: https://arxiv.org/abs/2602.03337 → 26, 84.

[IMS24] Florian Ingels, Camille Marchet, and Mikaël Salson. “On the number of k-mers admitting a given lexicographical minimizer”. In: *arXiv* (2024). DOI: 10.48550/arXiv.2412.17492. arXiv: 2412.17492 [cs.DS]. URL: https://arxiv.org/abs/2412.17492 → 26, 84.

[Ing+25] Florian Ingels, Lucas Robidou, Igor Martayan, Camille Marchet, and Antoine Limasset. “Minimizer Density revisited: Models and Multiminimizers”. In: *bioRxiv* (Nov. 2025). DOI: 10.1101/2025.11.21.689688 → 133.

[Int26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Combined Volumes 1–4. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Intel Corporation. 2026. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html → 34.

[IPB22] Luiz Irber, N. Tessa Pierce-Ward, and C. Titus Brown. “Sourmash Branchwater Enables Lightweight Petabyte-Scale Sequence Search”. In: *bioRxiv* (Nov. 2022). DOI: 10.1101/2022.11.02.514947. URL: http://dx.doi.org/10.1101/2022.11.02.514947 → 21.

[Irb+22] Luiz Irber, Phillip T. Brooks, Taylor Reiter, N. Tessa Pierce-Ward, Mahmudur Rahman Hera, David Koslicki, and C. Titus Brown. “Lightweight compositional analysis of metagenomes with FracMinHash and minimum metagenome covers”. In: *bioRxiv* (Jan. 2022). DOI: 10.1101/2022.01.11.475838. URL: http://dx.doi.org/10.1101/2022.01.11.475838 → 21, 145.

[ITM12] Zamin Iqbal, Isaac Turner, and Gil McVean. “High-throughput microbial population genomics using the Cortex variation assembler”. In: *Bioinformatics* 29.2 (Nov. 2012), pp. 275–276. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts673. URL: http://dx.doi.org/10.1093/bioinformatics/bts673 → 15.

[Jai+17] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. "A Fast Approximate Algorithm for Mapping Long Reads to Large Reference Databases". In: *Research in Computational Molecular Biology*. Springer International Publishing, 2017, pp. 66–81. ISBN: 9783319569703. DOI: 10.1007/978-3-319-56970-3_5. URL: http://dx.doi.org/10.1007/978-3-319-56970-3_5 → 157.

[Kar+11] Eric Karsenti, Silvia G. Acinas, Peer Bork, Chris Bowler, Colomban De Vargas, Jeroen Raes, Matthew Sullivan, Detlev Arendt, Francesca Benzoni, Jean-Michel Claverie, Mick Follows, Gaby Gorsky, Pascal Hingamp, Daniele Iudicone, Olivier Jaillon, Stefanie Kandels-Lewis, Uros Krzic, Fabrice Not, Hiroyuki Ogata, Stéphane Pesant, Emmanuel Georges Reynaud, Christian Sardet, Michael E. Sieracki, Sabrina Speich, Didier Velayoudon, Jean Weissenbach, and Patrick and Wincker. "A Holistic Approach to Marine Eco-Systems Biology". In: *PLoS Biology* 9.10 (Oct. 2011), e1001177. ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1001177. URL: http://dx.doi.org/10.1371/journal.pbio.1001177 → 17.

[Kar+25] Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Oleksandr Kulkov, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. "Efficient and accurate search in petabase-scale sequence repositories". In: *Nature* 647.8091 (Oct. 2025), pp. 1036–1044. ISSN: 1476-4687. DOI: 10.1038/s41586-025-09603-w. URL: http://dx.doi.org/10.1038/s41586-025-09603-w → 18.

[Kar09] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *50 Years of Integer Programming 1958-2008*. Springer Berlin Heidelberg, Nov. 2009, pp. 219–241. ISBN: 9783540682790. DOI: 10.1007/978-3-540-68279-0_8. URL: http://dx.doi.org/10.1007/978-3-540-68279-0_8 → 127.

[Kaz+22] Parham Kazemi, Johnathan Wong, Vladimir Nikolić, Hamid Mohamadi, René L Warren, and Inanç Birol. "ntHash2: recursive spaced seed hashing for nucleotide sequences". In: *Bioinformatics* 38.20 (Aug. 2022). Ed. by Peter Robinson, pp. 4812–4813. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btac564. URL: http://dx.doi.org/10.1093/bioinformatics/btac564 → 53.

[KDD17] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. "KMC 3: counting and manipulating k-mer statistics". In: *Bioinformatics* 33.17 (May 2017), pp. 2759–2761. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btx304. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/17/2759/49040995/bioinformatics_33_17_2759.pdf. URL: https://doi.org/10.1093/bioinformatics/btx304 → 28, 115, 127.

[Kil+24] Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J Treangen. "A near-tight lower bound on the density of forward sampling schemes". In: *Bioinformatics* 41.1 (Dec. 2024). Ed. by Yann Ponty. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae736. URL: http://dx.doi.org/10.1093/bioinformatics/btae736 → 129.

[KKDP22] Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. "Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2". In: *Genome Biology* 23.1 (Sept. 2022). ISSN: 1474-760X. DOI: 10.1186/s13059-022-02743-6. URL: http://dx.doi.org/10.1186/s13059-022-02743-6 → 17, 127.

[KMRK22] Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. "Lossless indexing with counting de Bruijn graphs". In: *Genome Research* 32.9 (May 2022), pp. 1754–1764. ISSN: 1549-5469. DOI: 10.1101/gr.276607.122. URL: http://dx.doi.org/10.1101/gr.276607.122 → 69.

[KPP26] Jamshed Khan, Rob Patro, and Prashant Pandey. “kache-hash: A dynamic, concurrent, and cache-efficient hash table for streaming k-mer operations”. In: *bioRxiv* (Feb. 2026). DOI: 10.64898/2026.02.13.705625. URL: http://dx.doi.org/10.64898/2026.02.13.705625 → 77.

[KR87] Richard M. Karp and Michael O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM Journal of Research and Development* 31.2 (Mar. 1987), pp. 249–260. ISSN: 0018-8646. DOI: 10.1147/rd.312.0249. URL: http://dx.doi.org/10.1147/rd.312.0249 → 49.

[KT05] Konstantinos T. Konstantinidis and James M. Tiedje. “Genomic insights that advance the species definition for prokaryotes”. In: *Proceedings of the National Academy of Sciences* 102.7 (Feb. 2005), pp. 2567–2572. ISSN: 1091-6490. DOI: 10.1073/pnas.0409727102. URL: http://dx.doi.org/10.1073/pnas.0409727102 → 19.

[KYLP19] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A. Pevzner. “Assembly of long, error-prone reads using repeat graphs”. In: *Nature Biotechnology* 37.5 (Apr. 2019), pp. 540–546. ISSN: 1546-1696. DOI: 10.1038/s41587-019-0072-8. URL: http://dx.doi.org/10.1038/s41587-019-0072-8 → 16.

[LBK15] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. “SIMD compression and the intersection of sorted integers”. In: *Software: Practice and Experience* 46.6 (Apr. 2015), pp. 723–749. ISSN: 1097-024X. DOI: 10.1002/spe.2326. URL: http://dx.doi.org/10.1002/spe.2326 → 102.

[Leh+26] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. “Modern Minimal Perfect Hashing: A Survey”. In: *ACM Computing Surveys* 58.10 (Mar. 2026), pp. 1–36. ISSN: 1557-7341. DOI: 10.1145/3797036. URL: http://dx.doi.org/10.1145/3797036 → 84.

[Lem17] Daniel Lemire. *Removing duplicates from lists quickly*. https://lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/. Apr. 2017. URL: https://lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/ → 62.

[Lev+26] Victor Levallois, Yoshihiro Shibuya, Bertrand Le Gal, Yoann Dufresne, Rob Patro, Pierre Peterlongo, and Giulio Ermanno Pibiri. “Kaminari: a frugal colored index for approximate k-mer queries”. In: *Bioinformatics Advances* (Apr. 2026). Ed. by Aida Ouangraoua. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbag120. URL: http://dx.doi.org/10.1093/bioadv/vbag120 → 28.

[Li+13] Yang Li, Pegah Kamousi, Fangqiu Han, Shengqi Yang, Xifeng Yan, and Subhash Suri. “Memory efficient minimum substring partitioning”. In: *Proceedings of the VLDB Endowment* 6.3 (Jan. 2013), pp. 169–180. ISSN: 2150-8097. DOI: 10.14778/2535569.2448951. URL: http://dx.doi.org/10.14778/2535569.2448951 → 27.

[Li09] Heng Li. *kseq*. https://github.com/attractivechaos/klib. 2009. URL: https://github.com/attractivechaos/klib → 38.

[Li16] Heng Li. “Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences”. In: *Bioinformatics* 32.14 (Mar. 2016), pp. 2103–2110. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw152. URL: http://dx.doi.org/10.1093/bioinformatics/btw152 → 28.

[Li18] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 34.18 (May 2018). Ed. by Inanc Birol, pp. 3094–3100. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/bty191. URL: http://dx.doi.org/10.1093/bioinformatics/bty191 → 28.

[Li20] Heng Li. *biofast*. https://github.com/lh3/biofast. 2020. URL: https://github.com/lh3/biofast → 38.

[LK10] Ping Li and Christian König. "b-Bit minwise hashing". In: *Proceedings of the 19th international conference on World wide web*. WWW '10. ACM, Apr. 2010, pp. 671–680. DOI: 10.1145/1772690.1772759. URL: http://dx.doi.org/10.1145/1772690.1772759 → 22.

[LKK19] Daniel Lemire, Owen Kaser, and Nathan Kurz. "Faster remainder by direct computation: Applications to compilers and software libraries". In: *Software: Practice and Experience* 49.6 (Feb. 2019), pp. 953–970. ISSN: 1097-024X. DOI: 10.1002/spe.2689. URL: http://dx.doi.org/10.1002/spe.2689 → 128.

[LKN13] V. Leis, Alfons Kemper, and T. Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2013, pp. 38–49. DOI: 10.1109/icde.2013.6544812. URL: http://dx.doi.org/10.1109/ICDE.2013.6544812 → 97.

[LL19] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (Oct. 2019), pp. 941–960. ISSN: 0949-877X. DOI: 10.1007/s00778-019-00578-5. URL: http://dx.doi.org/10.1007/s00778-019-00578-5 → 40.

[LMCP22] Téo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. "kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections". In: *Bioinformatics Advances* 2.1 (Jan. 2022). Ed. by Thomas Lengauer. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbac029. URL: http://dx.doi.org/10.1093/bioadv/vbac029 → 99.

[Lot97] M. Lothaire. *Combinatorics on words*. Vol. 17. Cambridge university press, 1997 → 89.

[LOZ12] Ping Li, Art Owen, and Cun-hui Zhang. "One Permutation Hashing". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/eaa32c96f620053cf442ad32258076b9-Paper.pdf → 21.

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets". en. In: vol. 75. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 25:1–25:16. DOI: 10.4230/LIPICS.SEA.2017.25. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2017.25 → 84.

[LY15] Yang Li and Xifeng Yan. "MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting". In: *arXiv* (2015). DOI: 10.48550/arXiv.1505.06550. arXiv: 1505.06550 [q-bio.GN]. URL: https://arxiv.org/abs/1505.06550 → 28.

[MABP25] Tommi Mäklin, Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. "Sequence alignment with k-bounded matching statistics". In: *bioRxiv* (May 2025). DOI: 10.1101/2025.05.19.654936. URL: http://dx.doi.org/10.1101/2025.05.19.654936 → 70, 87.

[Mar+17] Guillaume Marçais, David Pellow, Daniel Bork, Yaron Orenstein, Ron Shamir, and Carl Kingsford. "Improving the performance of minimizers and winnowing schemes". In: *Bioinformatics* 33.14 (July 2017), pp. i110–i117. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btx235. URL: http://dx.doi.org/10.1093/bioinformatics/btx235 → 26, 135.

[Mar+20] Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. "REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets". In: *Bioinformatics* 36.Supplement_1 (July 2020), pp. i177–i185. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaa487. URL: http://dx.doi.org/10.1093/bioinformatics/btaa487 → 17.

[Mar+23] Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. "Optimal gap-affine alignment in O(s) space". In: *Bioinformatics* 39.2 (Feb. 2023). Ed. by Pier Luigi Martelli. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btad074. URL: http://dx.doi.org/10.1093/bioinformatics/btad074 → 14.

[Mar+26] Igor Martayan, Léa Vandamme, Bede Constantinides, Bastien Cazaux, Charles Paperman, and Antoine Limasset. "Accelerating k-mer-based sequence filtering". In: *Peer Community Journal* 6 (June 2026). ISSN: 2804-3871. DOI: 10.24072/pcjournal.735. URL: http://dx.doi.org/10.24072/pcjournal.735 → 69.

[Mar24] Camille Marchet. "Advances in practical k-mer sets: essentials for the curious". In: *arXiv* (2024). DOI: 10.48550/arXiv.2409.05210. arXiv: 2409.05210 [q-bio.GN]. URL: https://arxiv.org/abs/2409.05210 → 83.

[MCLM24] Igor Martayan, Bastien Cazaux, Antoine Limasset, and Camille Marchet. "Conway-Bromage-Lyndon (CBL): an exact, dynamic representation of k-mer sets". In: *Bioinformatics* (June 2024). DOI: 10.1093/bioinformatics/btae217 → 89, 99, 107.

[MCVB16] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. "ntHash: recursive nucleotide hashing". In: *Bioinformatics* 32.22 (July 2016), pp. 3492–3494. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btw397. URL: http://dx.doi.org/10.1093/bioinformatics/btw397 → 51.

[MDK18] Guillaume Marçais, Dan DeBlasio, and Carl Kingsford. "Asymptotically optimal minimizers schemes". In: *Bioinformatics* 34.13 (June 2018), pp. i13–i22. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/bty258. URL: http://dx.doi.org/10.1093/bioinformatics/bty258 → 127, 129, 136.

[MEK24] Guillaume Marçais, C S Elder, and Carl Kingsford. "k-nonical space: sketching with reverse complements". In: *Bioinformatics* 40.11 (Oct. 2024), btae629. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae629 → 61.

[Mit01] M. Mitzenmacher. "The power of two choices in randomized load balancing". In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104. ISSN: 1045-9219. DOI: 10.1109/71.963420. URL: http://dx.doi.org/10.1109/71.963420 → 133.

[MK11] Guillaume Marçais and Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (Jan. 2011), pp. 764–770. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr011. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/6/764/48866141/bioinformatics_27_6_764.pdf. URL: https://doi.org/10.1093/bioinformatics/btr011 → 107, 115.

[MKL21] Camille Marchet, Mael Kerbiriou, and Antoine Limasset. "BLight: efficient exact associative structure for k-mers". In: *Bioinformatics* 37.18 (Apr. 2021), pp. 2858–2865. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab217. URL: https://doi.org/10.1093/bioinformatics/btab217 → 127.

[MLMP26] Igor Martayan, Loup Lobet, Camille Marchet, and Charles Paperman. "Helicase: Vectorized parsing and bitpacking of genomic sequences". In: *bioRxiv* (Mar. 2026). DOI: 10.64898/2026.03.19.712912 → 37.

[MMME20] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. "Fast gap-affine pairwise alignment using the wavefront algorithm". In: *Bioinformatics* 37.4 (Sept. 2020). Ed. by Peter Robinson, pp. 456–463. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaa777. URL: http://dx.doi.org/10.1093/bioinformatics/btaa777 → 14.

[MP71] Robert McNaughton and Seymour A Papert. "Counter-Free Automata". In: The MIT Press, 1971. URL: https://dl.acm.org/doi/abs/10.5555/1097043 → 40.

[MRSL25] Igor Martayan, Lucas Robidou, Yoshihiro Shibuya, and Antoine Limasset. "Hyper-k-mers: Efficient Streaming k-mers Representation". In: *Research in Computational Molecular Biology (RECOMB 2025)*. Springer Nature Switzerland, Apr. 2025. DOI: 10.1007/978-3-031-90252-9_33 → 111.

[MV20] Stefano Marchini and Sebastiano Vigna. "Compact Fenwick trees for dynamic ranking and selection". In: *Software: Practice and Experience* 50.7 (Jan. 2020), pp. 1184–1202. ISSN: 1097-024X. DOI: 10.1002/spe.2791. URL: http://dx.doi.org/10.1002/spe.2791 → 92.

[Mye23] Gene Myers. *FASTK: A fast K-mer counter for high-fidelity shotgun datasets*. https://github.com/thegenemyers/FASTK. 2023. URL: https://github.com/thegenemyers/FASTK → 115.

[Mye99] Gene Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM* 46.3 (May 1999), pp. 395–415. ISSN: 1557-735X. DOI: 10.1145/316542.316550. URL: http://dx.doi.org/10.1145/316542.316550 → 41.

[Myk72] Johannes Mykkeltveit. "A proof of Golomb's conjecture for the de Bruijn graph". In: *Journal of Combinatorial Theory, Series B* 13.1 (Aug. 1972), pp. 40–45. ISSN: 0095-8956. DOI: 10.1016/0095-8956(72)90006-8. URL: http://dx.doi.org/10.1016/0095-8956(72)90006-8 → 127.

[Ndi+24] Malick Ndiaye, Silvia Prieto-Baños, Lucy M. Fitzgerald, Ali Yazdizadeh Kharrazi, Sergey Oreshkov, Christophe Dessimoz, Fritz J. Sedlazeck, Natasha Glover, and Sina Majidian. "When less is more: sketching with minimizers in genomics". In: *Genome Biology* 25.1 (Oct. 2024). ISSN: 1474-760X. DOI: 10.1186/s13059-024-03414-4. URL: http://dx.doi.org/10.1186/s13059-024-03414-4 → 25.

[Nun+23] Igor Nunes, Mike Heddes, Pere Vergés, Danny Abraham, Alex Veidenbaum, Alex Nicolau, and Tony Givargis. "DotHash: Estimating Set Similarity Metrics for Link Prediction and Document Deduplication". In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '23. ACM, Aug. 2023, pp. 1758–1769. DOI: 10.1145/3580305.3599314. URL: http://dx.doi.org/10.1145/3580305.3599314 → 23, 158.

[Nur+22] Sergey Nurk et al. "The complete sequence of a human genome". In: *Science* 376.6588 (Apr. 2022), pp. 44–53. ISSN: 1095-9203. DOI: 10.1126/science.abj6987. URL: http://dx.doi.org/10.1126/science.abj6987 → 68.

[NW70] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (Mar. 1970), pp. 443–453. ISSN: 0022-2836. DOI: 10.1016/0022-2836(70)90057-4. URL: http://dx.doi.org/10.1016/0022-2836(70)90057-4 → 14.

[Ond+16] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. "Mash: fast genome and metagenome distance estimation using MinHash". In: *Genome Biology* 17.1 (June 2016). ISSN: 1474-760X. DOI: 10.1186/s13059-016-0997-x. URL: http://dx.doi.org/10.1186/s13059-016-0997-x → 20, 21, 145.

[One19] One Codex. *needletail*. https://github.com/onecodex/needletail. 2019. URL: https://github.com/onecodex/needletail → 38, 39, 44.

[Ore+16] Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. "Compact Universal k-mer Hitting Sets". In: *Algorithms in Bioinformatics*. Springer International Publishing, 2016, pp. 257–268. ISBN: 9783319436814. DOI: 10.1007/978-3-319-43681-4_21. URL: http://dx.doi.org/10.1007/978-3-319-43681-4_21 → 127, 139.

[Ore+17] Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. "Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing". In: *PLOS Computational Biology* 13.10 (Oct. 2017). Ed. by Benjamin J. Raphael, e1005777. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1005777. URL: http://dx.doi.org/10.1371/journal.pcbi.1005777 → 127, 139.

[Pan+18] Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. "Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index". In: *Cell Systems* 7.2 (Aug. 2018), 201–207.e4. ISSN: 2405-4712. DOI: 10.1016/j.cels.2018.05.021. URL: http://dx.doi.org/10.1016/j.cels.2018.05.021 → 18.

[Pat+25] Rob Patro, Siddhant Bharti, Prajwal Singhania, Rakrish Dhakal, Thomas J. Dahlstrom, and Ragnar Groot Koerkamp. "mim: A lightweight auxiliary index to enable fast, parallel, gzipped FASTQ parsing". In: *bioRxiv* (Nov. 2025). DOI: 10.1101/2025.11.24.690271. URL: http://dx.doi.org/10.1101/2025.11.24.690271 → 38, 78.

[Pel+23] David Pellow, Lianrong Pu, Barış Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein. "Efficient minimizer orders for large values of k using minimum decycling sets". In: *Genome Research* (Aug. 2023). ISSN: 1549-5469. DOI: 10.1101/gr.277644.123. URL: http://dx.doi.org/10.1101/gr.277644.123 → 78, 128, 151.

[Pen17] Elizabeth Pennisi. "Biologists propose to sequence the DNA of all life on Earth". In: *Science* (Feb. 2017). ISSN: 1095-9203. DOI: 10.1126/science.aal0824. URL: http://dx.doi.org/10.1126/science.aal0824 → 47.

[Pib22] Giulio Ermanno Pibiri. "Sparse and skew hashing of K-mers". In: *Bioinformatics* 38.Supplement_1 (June 2022), pp. i185–i194. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btac245. URL: https://doi.org/10.1093/bioinformatics/btac245 → 64.

[Pie+19] N. Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C. Titus Brown. "Large-scale sequence comparisons with sourmash". In: *F1000Research* 8 (July 2019), p. 1006. ISSN: 2046-1402. DOI: 10.12688/f1000research.19675.1. URL: http://dx.doi.org/10.12688/f1000research.19675.1 → 21, 145.

[PK21] Giulio Ermanno Pibiri and Shunsuke Kanda. "Rank/select queries over mutable bitmaps". In: *Information Systems* 99 (July 2021), p. 101756. ISSN: 0306-4379. DOI: 10.1016/j.is.2021.101756. URL: http://dx.doi.org/10.1016/j.is.2021.101756 → 92.

[PP26] Giulio Ermanno Pibiri and Rob Patro. “Optimizing sparse and skew hashing: faster k-mer dictionaries”. In: *bioRxiv* (Jan. 2026). DOI: 10.64898/2026.01.21.700884. URL: http://dx.doi.org/10.64898/2026.01.21.700884 → 83.

[PR24] Chenxu Pan and Knut Reinert. “A simple refined DNA minimizer operator enables 2-fold faster computation”. In: *Bioinformatics* 40.2 (Jan. 2024), btae045. DOI: 10.1093/bioinformatics/btae045. URL: https://doi.org/10.1093/bioinformatics/btae045 → 61.

[PSL23] Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset. “Locality-preserving minimal perfect hashing of k-mers”. In: *Bioinformatics* 39.Supplement_1 (June 2023), pp. i534–i543. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btad219. URL: http://dx.doi.org/10.1093/bioinformatics/btad219 → 83.

[PSS23] Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin. “An Algebraic Approach to Vectorial Programs”. en. In: vol. 254. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 51:1–51:23. DOI: 10.4230/LIPICS.STACS.2023.51. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2023.51 → 40, 47.

[PT21] Giulio Ermanno Pibiri and Roberto Trani. “PTHash: Revisiting FCH Minimal Perfect Hashing”. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’21. ACM, July 2021, pp. 1339–1348. DOI: 10.1145/3404835.3462849. URL: http://dx.doi.org/10.1145/3404835.3462849 → 84.

[PV17] Giulio Ermanno Pibiri and Rossano Venturini. “Dynamic Elias-Fano Representation”. en. In: vol. 78. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 30:1–30:14. DOI: 10.4230/LIPICS.CPM.2017.30. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2017.30 → 94.

[PWW97] Alex Peleg, Sam Wilkie, and Uri Weiser. “Intel MMX for multimedia PCs”. In: *Communications of the ACM* 40.1 (Jan. 1997), pp. 24–38. ISSN: 1557-7317. DOI: 10.1145/242857.242865. URL: http://dx.doi.org/10.1145/242857.242865 → 33.

[RCL25] Timothé Rouzé, Rayan Chikhi, and Antoine Limasset. “Inverted colored de Bruijn Graph for practical kmer sets storage”. In: *bioRxiv* (Dec. 2025). DOI: 10.64898/2025.12.08.692073. URL: http://dx.doi.org/10.64898/2025.12.08.692073 → 18.

[RM21] Amatur Rahman and Paul Medevedev. “Representation of k-mer Sets Using Spectrum-Preserving String Sets”. In: *Journal of Computational Biology* 28.4 (Apr. 2021), pp. 381–394. ISSN: 1557-8666. DOI: 10.1089/cmb.2020.0431. URL: http://dx.doi.org/10.1089/cmb.2020.0431 → 16.

[RMML23] Timothé Rouzé, Igor Martayan, Camille Marchet, and Antoine Limasset. “Fractional Hitting Sets for Efficient and Lightweight Genomic Data Sketching”. In: *23rd International Workshop on Algorithms in Bioinformatics (WABI 2023)*. Vol. 273. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Aug. 2023. DOI: 10.4230/LIPIcs.WABI.2023.15 → 145.

[RMML25] Timothé Rouzé, Igor Martayan, Camille Marchet, and Antoine Limasset. “Fractional hitting sets for efficient multiset sketching”. In: *Algorithms for Molecular Biology* 20.1 (Feb. 2025), p. 1. DOI: 10.1186/s13015-024-00268-0 → 145.

[Rob+04] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. “Reducing storage requirements for biological sequence comparison”. In: *Bioinformatics* 20.18 (July 2004), pp. 3363–3369. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bth408. URL: https://doi.org/10.1093/bioinformatics/bth408 → 25, 135.

[Row19] Will P. M. Rowe. “When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data”. In: *Genome Biology* 20.1 (Sept. 2019). ISSN: 1474-760X. DOI: 10.1186/s13059-019-1809-x. URL: http://dx.doi.org/10.1186/s13059-019-1809-x → 19, 21.

[RPK23] Mahmudur Rahman Hera, N. Tessa Pierce-Ward, and David Koslicki. “Deriving confidence intervals for mutation rates across a wide range of evolutionary distances using FracMinHash”. In: *Genome Research* (June 2023). ISSN: 1549-5469. DOI: 10.1101/gr.277651.123. URL: http://dx.doi.org/10.1101/gr.277651.123 → 21.

[Rus78] Richard M. Russell. “The Cray-1 Computer System”. In: *Commun. ACM* 21.1 (1978), pp. 63–72. DOI: 10.1145/359327.359336. URL: https://doi.org/10.1145/359327.359336 → 33.

[SA23] Sebastian Schmidt and Jarno N. Alanko. “Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time”. In: *Algorithms for Molecular Biology* 18.1 (July 2023). ISSN: 1748-7188. DOI: 10.1186/s13015-023-00227-1. URL: http://dx.doi.org/10.1186/s13015-023-00227-1 → 17.

[SA24] Sergey A. Shiryev and Richa Agarwala. “Indexing and searching petabase-scale nucleotide resources”. In: *Nature Methods* 21.6 (May 2024), pp. 994–1002. ISSN: 1548-7105. DOI: 10.1038/s41592-024-02280-z. URL: http://dx.doi.org/10.1038/s41592-024-02280-z → 17.

[Sah21] Kristoffer Sahlin. “Effective sequence similarity detection with strobemers”. In: *Genome Research* 31.11 (Oct. 2021), pp. 2080–2094. ISSN: 1549-5469. DOI: 10.1101/gr.275648.121. URL: http://dx.doi.org/10.1101/gr.275648.121 → 28.

[Sah22] Kristoffer Sahlin. “Strobealign: flexible seed size enables ultra-fast and accurate read alignment”. In: *Genome Biology* 23.1 (Dec. 2022). ISSN: 1474-760X. DOI: 10.1186/s13059-022-02831-7. URL: http://dx.doi.org/10.1186/s13059-022-02831-7 → 28.

[SBCM23] Kristoffer Sahlin, Thomas Baudeau, Bastien Cazaux, and Camille Marchet. “A survey of mapping algorithms in the long-reads era”. In: *Genome Biology* 24.1 (June 2023). ISSN: 1474-760X. DOI: 10.1186/s13059-023-02972-3. URL: http://dx.doi.org/10.1186/s13059-023-02972-3 → 28.

[SBK22] Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. “Efficient Reconciliation of Genomic Datasets of High Similarity”. en. In: *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 14:1–14:14. DOI: 10.4230/LIPIcs.WABI.2022.14. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2022.14 (visited on 10/24/2024) → 99, 111.

[Sch+23] Sebastian Schmidt, Shahbaz Khan, Jarno N. Alanko, Giulio E. Pibiri, and Alexandru I. Tomescu. “Matchtigs: minimum plain text representation of k-mer sets”. In: *Genome Biology* 24.1 (June 2023). ISSN: 1474-760X. DOI: 10.1186/s13059-023-02968-z. URL: http://dx.doi.org/10.1186/s13059-023-02968-z → 17.

[Sch+24] Manfred Schartl, Joost M. Woltering, Iker Irisarri, Kang Du, Susanne Kneitz, Martin Pippel, Thomas Brown, Paolo Franchini, Jing Li, Ming Li, Mateus Adolfi, Sylke Winkler, Josane de Freitas Sousa, Zhuoxin Chen, Sandra Jacinto, Evgeny Z. Kvon, Luis Rogério Correa de Oliveira, Erika Monteiro, Danielson Baia Amaral, Thorsten Burmester, Domitille Chalopin, Alexander Suh, Eugene Myers, Oleg Simakov, Igor Schneider, and Axel Meyer. “The genomes of all lungfish inform on genome expansion and tetrapod evolution”. In: *Nature* 634.8032 (Aug. 2024), pp. 96–103. ISSN: 1476-4687. DOI: 10.1038/s41586-024-07830-1. URL: http://dx.doi.org/10.1038/s41586-024-07830-1 → 47.

[Ser+22] Mantas Sereika, Rasmus Hansen Kirkegaard, Søren Michael Karst, Thomas Yssing Michaelsen, Emil Aarre Sørensen, Rasmus Dam Wollenberg, and Mads Albertsen. "Oxford Nanopore R10.4 long-read sequencing enables the generation of near-finished bacterial genomes from pure cultures and metagenomes without short-read or reference polishing". In: *Nature Methods* 19.7 (July 2022), pp. 823–826. ISSN: 1548-7105. DOI: 10.1038/s41592-022-01539-7. URL: http://dx.doi.org/10.1038/s41592-022-01539-7 → 111.

[Ser04] Olivier Serre. "Vectorial languages and linear temporal logic". In: *Theor. Comput. Sci.* 310.1-3 (2004), pp. 79–116. DOI: 10.1016/S0304-3975(03)00346-3. URL: https://doi.org/10.1016/S0304-3975(03)00346-3 → 40, 47.

[SK16] Brad Solomon and Carl Kingsford. "Fast search of thousands of short-read sequencing experiments". In: *Nature Biotechnology* 34.3 (Feb. 2016), pp. 300–302. ISSN: 1546-1696. DOI: 10.1038/nbt.3442. URL: http://dx.doi.org/10.1038/nbt.3442 → 17.

[SL14] Anshumali Shrivastava and Ping Li. "Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, June 2014, pp. 557–565. URL: https://proceedings.mlr.press/v32/shrivastava14.html → 22.

[SLI25] Wei Shen, John A. Lees, and Zamin Iqbal. "Efficient sequence alignment against millions of prokaryotic genomes with LexicMap". In: *Nature Biotechnology* (Sept. 2025). ISSN: 1546-1696. DOI: 10.1038/s41587-025-02812-8. URL: http://dx.doi.org/10.1038/s41587-025-02812-8 → 157.

[SLLH16] Wei Shen, Shuai Le, Yan Li, and Fuquan Hu. "SeqKit: A Cross-Platform and Ultrafast Toolkit for FASTA/Q File Manipulation". In: *PLOS ONE* 11.10 (Oct. 2016). Ed. by Quan Zou, e0163962. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0163962. URL: http://dx.doi.org/10.1371/journal.pone.0163962 → 69, 72.

[SMLD24] Caleb Smith, Igor Martayan, Antoine Limasset, and Yoann Dufresne. "Brisk: Exact resource-efficient dictionary for k-mers". In: *bioRxiv* (Nov. 2024). DOI: 10.1101/2024.11.26.625346 → 103.

[SNC77] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences* 74.12 (Dec. 1977), pp. 5463–5467. ISSN: 1091-6490. DOI: 10.1073/pnas.74.12.5463. URL: http://dx.doi.org/10.1073/pnas.74.12.5463 → 13.

[Soy23] Claire Soyez-Martin. "From semigroup theory to vectorization: recognizing regular languages." PhD thesis. University of Lille, France, 2023. DOI: 10.70675/09db0f47z19fbz4c84zbeedza5bfb8f585c8. URL: http://dx.doi.org/10.70675/09db0f47z19fbz4c84zbeedza5bfb8f585c8 → 40.

[SSZ24] Wei Shen, Botond Sipos, and Liuyang Zhao. "SeqKit2: A Swiss army knife for sequence and alignment processing". In: *iMeta* 3.3 (Apr. 2024). ISSN: 2770-596X. DOI: 10.1002/imt2.191. URL: http://dx.doi.org/10.1002/imt2.191 → 69, 72.

[Ste+15] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. "Big Data: Astronomical or Genomical?" In: *PLOS Biology* 13.7 (July 2015), e1002195. ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1002195. URL: http://dx.doi.org/10.1371/journal.pbio.1002195 → 17.

[STO26] Arseny Shur, Ido Tziony, and Yaron Orenstein. “10-minimizers: a promising class of constant-space minimizers”. In: *bioRxiv* (Mar. 2026). DOI: 10.64898/2026.03.16.712052. URL: http://dx.doi.org/10.64898/2026.03.16.712052 → 78, 130.

[SVB23] Ondřej Sladký, Pavel Veselý, and Karel Břinda. “Masked superstrings as a unified framework for textual k-mer set representations”. In: *bioRxiv* (Feb. 2023). DOI: 10.1101/2023.02.01.526717. URL: http://dx.doi.org/10.1101/2023.02.01.526717 → 83, 87.

[SVB24] Ondřej Sladký, Pavel Veselý, and Karel Břinda. “Towards Efficient k-Mer Set Operations via Function-Assigned Masked Superstrings”. In: *bioRxiv* (Mar. 2024). DOI: 10.1101/2024.03.06.583483. URL: http://dx.doi.org/10.1101/2024.03.06.583483 → 83, 88.

[SVB25] Ondřej Sladký, Pavel Veselý, and Karel Břinda. “FroM Superstring to Indexing: a space-efficient index for unconstrained k-mer sets using the Masked Burrows-Wheeler Transform (MBWT)”. In: *Bioinformatics Advances* 6.1 (Nov. 2025). Ed. by Thomas Lengauer. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbaf290. URL: http://dx.doi.org/10.1093/bioadv/vbaf290 → 83, 87, 88.

[SW17] Joe Sawada and Aaron Williams. “Practical algorithms to rank necklaces, Lyndon words, and de Bruijn sequences”. In: *Journal of Discrete Algorithms* 43 (Mar. 2017), pp. 95–110. ISSN: 1570-8667. DOI: 10.1016/j.jda.2017.01.003. URL: http://dx.doi.org/10.1016/j.jda.2017.01.003 → 90, 96.

[SW81] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (Mar. 1981), pp. 195–197. ISSN: 0022-2836. DOI: 10.1016/0022-2836(81)90087-5. URL: http://dx.doi.org/10.1016/0022-2836(81)90087-5 → 14.

[SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. “Winnowing: local algorithms for document fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. SIGMOD ’03. New York, NY, USA: Association for Computing Machinery, June 2003, pp. 76–85. ISBN: 9781581136340. DOI: 10.1145/872757.872770. URL: https://dl.acm.org/doi/10.1145/872757.872770 (visited on 10/18/2024) → 25.

[SY21] Jim Shaw and Yun William Yu. “Theory of local k-mer selection with applications to long-read alignment”. In: *Bioinformatics* 38.20 (Nov. 2021). Ed. by Can Alkan, pp. 4659–4669. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btab790. URL: http://dx.doi.org/10.1093/bioinformatics/btab790 → 27, 28.

[TD25] Noam Teyssier and Alexander Dobin. “BINSEQ: A Family of High-Performance Binary Formats for Nucleotide Sequences”. In: *bioRxiv* (Apr. 2025). DOI: 10.1101/2025.04.08.647863. URL: http://dx.doi.org/10.1101/2025.04.08.647863 → 78.

[Tey25] Noam Teyssier. *paraseq*. https://github.com/noamteyssier/paraseq. 2025. URL: https://github.com/noamteyssier/paraseq → 44.

[TKPP18] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. “Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation”. In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2018, pp. 34–41. URL: https://adms-conf.org/2018-camera-ready/SIMDWindowPaper_ADMS%2718.pdf → 59.

[Val26] Valve Corporation. *Steam Hardware & Software Survey: March 2026*. https://store.steampowered.com/hwsurvey/. Mar. 2026. URL: https://store.steampowered.com/hwsurvey/ → 34.

[VCL25] Léa Vandamme, Bastien Cazaux, and Antoine Limasset. "K2R: Tinted de Bruijn Graphs implementation for efficient read extraction from sequencing datasets". In: *Bioinformatics Advances* (May 2025), vbaf111. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbaf111 → 69.

[Vig08] Sebastiano Vigna. "Broadword Implementation of Rank/Select Queries". In: *Experimental Algorithms*. Springer Berlin Heidelberg, 2008, pp. 154–168. ISBN: 9783540685524. DOI: 10.1007/978-3-540-68552-4_12. URL: http://dx.doi.org/10.1007/978-3-540-68552-4_12 → 92.

[Wal44] Abraham Wald. "On Cumulative Sums of Random Variables". In: *The Annals of Mathematical Statistics* 15.3 (Sept. 1944), pp. 283–296. ISSN: 0003-4851. DOI: 10.1214/aoms/1177731235. URL: http://dx.doi.org/10.1214/aoms/1177731235 → 197.

[Wan+19] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. "Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 631–648. URL: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang → 69, 72.

[Wit23] Roland Wittler. "General encoding of canonical k-mers". In: *Peer Community Journal* 3 (Sept. 2023). ISSN: 2804-3871. DOI: 10.24072/pcjournal.323. URL: http://dx.doi.org/10.24072/pcjournal.323 → 90.

[WLL19] Derrick E Wood, Jennifer Lu, and Ben Langmead. "Improved metagenomic analysis with Kraken 2". In: *Genome biology* 20 (2019), pp. 1–13. DOI: 10.1186/s13059-019-1891-0 → 28, 64.

[WS14] Derrick E Wood and Steven L Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biology* 15.3 (Mar. 2014). ISSN: 1474-760X. DOI: 10.1186/gb-2014-15-3-r46. URL: http://dx.doi.org/10.1186/gb-2014-15-3-r46 → 28.

[Xu+24] Weihong Xu, Po-Kai Hsu, Niema Moshiri, Shimeng Yu, and Tajana Rosing. "HyperGen: compact and efficient genome sketching using hyperdimensional vectors". In: *Bioinformatics* 40.7 (July 2024). Ed. by Can Alkan. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae452. URL: http://dx.doi.org/10.1093/bioinformatics/btae452 → 23.

[YW20] Yun William Yu and Griffin M. Weber. "HyperMinHash: MinHash in LogLog space". In: *IEEE Transactions on Knowledge and Data Engineering* (2020), pp. 1–1. ISSN: 2326-3865. DOI: 10.1109/tkde.2020.2981311. URL: http://dx.doi.org/10.1109/TKDE.2020.2981311 → 22.

[Zak+24] Mohsen Zakeri, Nathaniel K. Brown, Omar Y. Ahmed, Travis Gagie, and Ben Langmead. "Movi: A fast and cache-efficient full-text pangenome index". In: *iScience* 27.12 (Dec. 2024), p. 111464. ISSN: 2589-0042. DOI: 10.1016/j.isci.2024.111464. URL: http://dx.doi.org/10.1016/j.isci.2024.111464 → 123.

[ZAK13] Dong Zhou, David G. Andersen, and Michael Kaminsky. "Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences". In: *Experimental Algorithms*. Springer Berlin Heidelberg, 2013, pp. 151–163. ISBN: 9783642385278. DOI: 10.1007/978-3-642-38527-8_15. URL: http://dx.doi.org/10.1007/978-3-642-38527-8_15 → 92.

[ZBGL26] Mohsen Zakeri, Nathaniel K Brown, Travis Gagie, and Ben Langmead. “Movi 2: Fast and Space-Efficient Queries on Pangenomes”. In: *Bioinformatics* (June 2026). Ed. by Macha Nikolski. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btag362. URL: http://dx.doi.org/10.1093/bioinformatics/btag362 → 123.

[Zha+23] Hao Zhang, Honglei Song, Xiaoming Xu, Qixin Chang, Mingkai Wang, Yanjie Wei, Zekun Yin, Bertil Schmidt, and Weiguo Liu. “RabbitFX: Efficient Framework for FASTA/Q File Parsing on Modern Multi-Core Platforms”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 20.3 (May 2023), pp. 2341–2348. ISSN: 2374-0043. DOI: 10.1109/tcbb.2022.3219114. URL: http://dx.doi.org/10.1109/TCBB.2022.3219114 → 38, 39.

[Zha19] XiaoFei Zhao. “BinDash, software for fast genome distance estimation on a typical personal laptop”. In: *Bioinformatics* 35.4 (Feb. 2019), pp. 671–673. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty651. URL: https://doi.org/10.1093/bioinformatics/bty651 → 22.

[ZKM20] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. “Improved design and analysis of practical minimizers”. In: *Bioinformatics* 36.Supplement_1 (July 2020), pp. i119–i127. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaa472. URL: http://dx.doi.org/10.1093/bioinformatics/btaa472 → 26.

[ZKM21] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. “Sequence-specific minimizers via polar sets”. In: *Bioinformatics* 37.Supplement_1 (July 2021), pp. i187–i195. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btab313. URL: http://dx.doi.org/10.1093/bioinformatics/btab313 → 134, 135.

[ZMK23] Hongyu Zheng, Guillaume Marçais, and Carl Kingsford. “Creating and Using Minimizer Sketches in Computational Genomics”. In: *Journal of Computational Biology* 30.12 (Dec. 2023), pp. 1251–1276. ISSN: 1557-8666. DOI: 10.1089/cmb.2023.0094. URL: http://dx.doi.org/10.1089/cmb.2023.0094 → 26.

[ZSR25] Jens Zentgraf, Johanna Elena Schmitz, and Sven Rahmann. “Cleanifier: contamination removal from microbial sequences using spaced seeds of a human pangenome index”. In: *Bioinformatics* 42.1 (Nov. 2025). Ed. by Inanc Birol. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaf632. URL: http://dx.doi.org/10.1093/bioinformatics/btaf632 → 69, 72.

A. A forward scheme for canonical minimizers

As discussed in Section 8.3, the canonical minimizer scheme presented there picks the leftmost minimizer on the canonical strand and the rightmost on the reverse one, so a strand flip between adjacent windows can make the selected position jump backward. The natural fix is to break ties at the *innermost* position instead, which is invariant under reverse complement and therefore avoids the strand-dependent jump. The downside is that finding the innermost minimum is slightly more involved than the leftmost or rightmost one, and it requires w to be odd so that the window has a well-defined center. This scheme is currently implemented in the minimizer-iter crate, though not especially optimized at the moment.

A.1. Innermost tie-breaking

As in Section 7.2.3, we assign each k-mer a canonical hash $\widehat{H}(x) = H(x) + H(\mathrm{rc}(x))$ so that a k-mer and its reverse complement always have the same hash. If the minimum canonical hash in a window is achieved at a unique position, that position is selected and no tie-breaking is needed. The challenge is when multiple positions share the minimum hash.

The main idea is to always select the *innermost* minimizer, i.e., the position closest to the center $c = (w-1)/2$ of the window. When two or more positions achieve the minimum hash, we pick the one minimizing $|p - c|$. When exactly two positions p and $w-1-p$ are equidistant from c (a symmetric tie), we break the tie using the canonical strand of the window, determined as in Section 8.3 (e.g. via the GT count): if the window is canonical we select the leftmost position p, otherwise we select the rightmost $w-1-p$.

This rule is canonical by construction. The symmetry $\widehat{H}(x) = \widehat{H}(\mathrm{rc}(x))$ ensures that the minimum positions of $\overline{W}$ are the mirrors $\{w-1-p\}$ of those of W, and mirroring preserves which position is innermost. For a symmetric tie, the canonical strand of $\overline{W}$ is by definition the reverse of that of W, swapping the leftmost/rightmost tie-break so that the mirror position is selected. We prove in Section A.2 that the rule also yields a forward scheme.

Table A.1.: Forward canonical minimizer selection for $w = 7$ ($c = 3$) on the hash sequence $[18, 42, 71, 15, 37, 82, 58, 15, 64, 15, 39, 92, 23, 45]$. Minimizers are in bold and the selected minimizer is highlighted. The absolute positions selected are non-decreasing.

Window	0	1	2	3	4	5	6	Case	Absolute pos.
W_0	18	42	71	**15**	37	82	58	unique min	3
W_1	42	71	**15**	37	82	58	**15**	innermost min	3
W_2	71	**15**	37	82	58	**15**	64	symmetric tie	3 or 7
W_3	**15**	37	82	58	**15**	64	**15**	innermost min	7
W_4	37	82	58	**15**	64	**15**	39	innermost min	7
W_5	82	58	**15**	64	**15**	39	92	symmetric tie	7 or 9
W_6	58	**15**	64	**15**	39	92	23	innermost min	9
W_7	**15**	64	**15**	39	92	23	45	innermost min	9

We walk through the example in Table A.1 window by window. W_0 has a unique minimum at position 3, the center of the window. W_1 has an asymmetric tie between positions 2 and 6 (distances 1 and 3 from center), and position 2 wins outright. W_2 has a symmetric tie

between positions 1 and 5, both at distance 2 from center, and the canonical strand of the window determines which is selected. Regardless of this choice, when the window shifts the right candidate moves from position 5 to position 4 in W_3. There, hash 15 appears at three positions (0, 4, 6), but position 4 is the unique innermost at distance $|4-3|=1$, so it is selected without a tie-break (absolute 7). W_5 has a second symmetric tie at positions 2 and 4, again resolved by the canonical strand. In W_6 and W_7, the minimum hash appears twice but one occurrence is strictly closer to center and wins without a strand-based tie-break.

A.2. Proof of forward property

Let p be the selected relative position in W_i and $i+p$ the selected absolute position. We show the selected absolute position in W_{i+1} is at least $i+p$.

If $p=0$, the selected k-mer is absent from W_{i+1}, and every position in W_{i+1} has absolute coordinate at least $i+1>i+p$.

If $p\geq 1$, the selected k-mer appears at relative position $p-1$ in W_{i+1} with absolute coordinate $i+p$, and still achieves the minimum hash. The selected position in W_{i+1} is the innermost among all tied positions, so we only need to show that no tied position with absolute coordinate less than $i+p$ can be that innermost one.

Any such position sits at relative $r\leq p-2$ in W_{i+1}, corresponding to relative $r+1\leq p-1$ in W_i. If $r+1<c$ (left of center), its distance from center in W_i is $c-(r+1)$, and in W_{i+1} it becomes $c-r=c-(r+1)+1$, strictly larger. Since p was innermost in W_i, $|p-c|\leq c-(r+1)$, so $|p-1-c|\leq |p-c|+1\leq c-r$, which means position r is no closer to center than $p-1$ in W_{i+1}. If $r+1\geq c$ (right of center) with $r+1<p$, its distance from center in W_i is $r+1-c<p-c=|p-c|$, contradicting p being innermost in W_i. This case cannot occur.

Thus, no tied position at absolute $<i+p$ is innermost in W_{i+1}, so the selected absolute position is at least $i+p$.

B. Experiments on parsing

B.1. Features of the benchmarked CPUs

Table B.1.: Instruction-set extensions supported by the benchmarked CPUs.

CPU	Year	SSE	AVX2	BMI2	NEON
Intel					
Xeon X5670	2010	+	–	–	–
Xeon E5-2620	2012	+	–	–	–
Xeon E7-4850 V3	2015	+	+	+	–
Xeon E5-2620 V4	2016	+	+	+	–
Xeon Gold 6130	2017	+	+	+	–
Xeon Gold 5218	2019	+	+	+	–
Xeon Gold 5318Y	2021	+	+	+	–
Xeon Silver 4314	2021	+	+	+	–
Xeon Gold 6442Y	2023	+	+	+	–
Core Ultra 7 165H *[1]	2023	+	+	+	–
AMD					
Epyc 7301	2017	+	+	~[2]	–
Epyc 7452	2019	+	+	~	–
Epyc 7642	2019	+	+	~	–
Epyc 7513	2021	+	+	+	–
Epyc 9254	2022	+	+	+	–
Ryzen 5 8500G *	2024	+	+	+	–
ARM					
Apple M1 *	2020	–	–	–	+
Apple M3 Pro *	2023	–	–	–	+
Neoverse-V2	2023	–	–	–	+

B.2. Additional experiments on data read from disk

B.3. Additional experiments on data loaded in RAM

B.3.1. Throughput

B.3.2. Instructions and cycles

B.3.3. Branches and branch misses

[1] (*) personal computers, not benchmarked in a reproducible environment
[2] (~) microcoded PDEP/PEXT instructions (very slow)

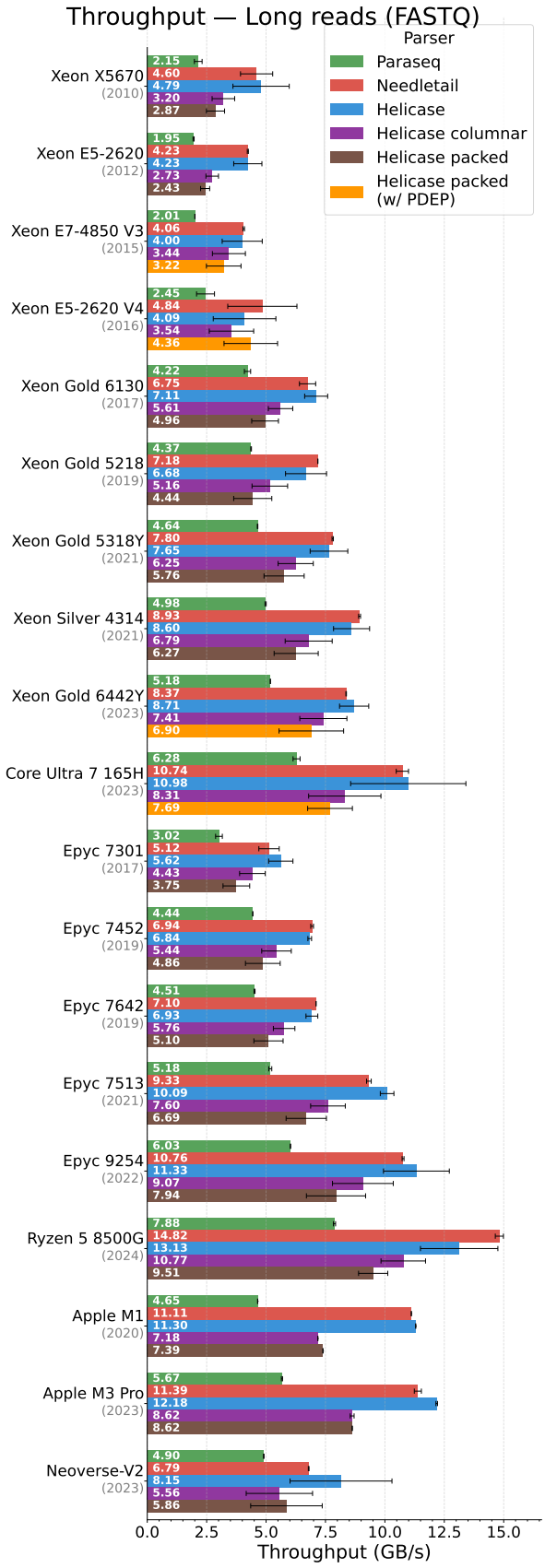
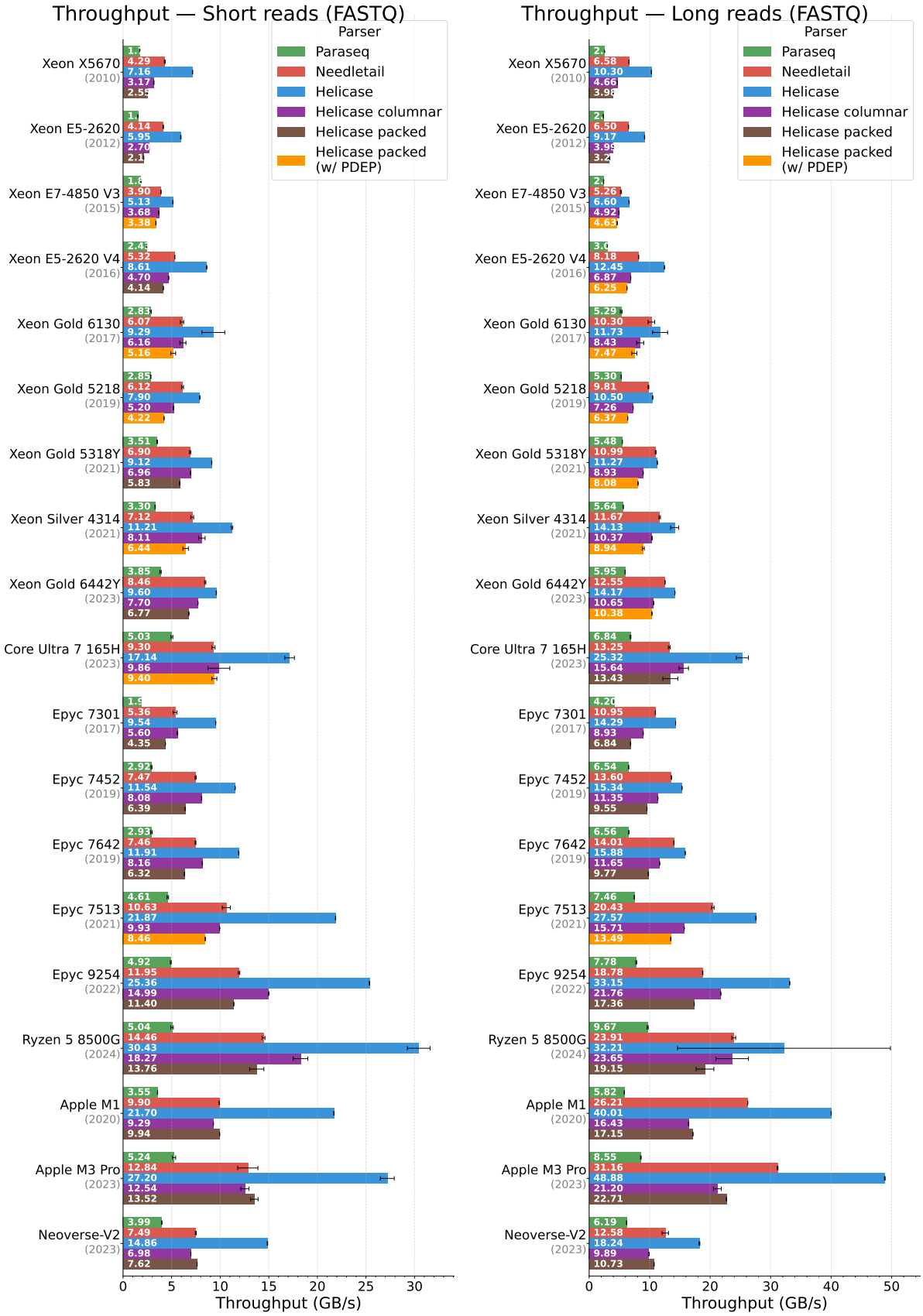


Figure B.1.: Throughput of each parser for long reads on multiple CPUs, sorted by manufacturer and year.



(a) On short reads (FASTQ format).

(b) On long reads (FASTQ format).

Figure B.2.: Throughput of each parser for data loaded in RAM on multiple CPUs, sorted by manufacturer and year. Needletail and Paraseq both have to use a reader over a slice, which degrades their performance.

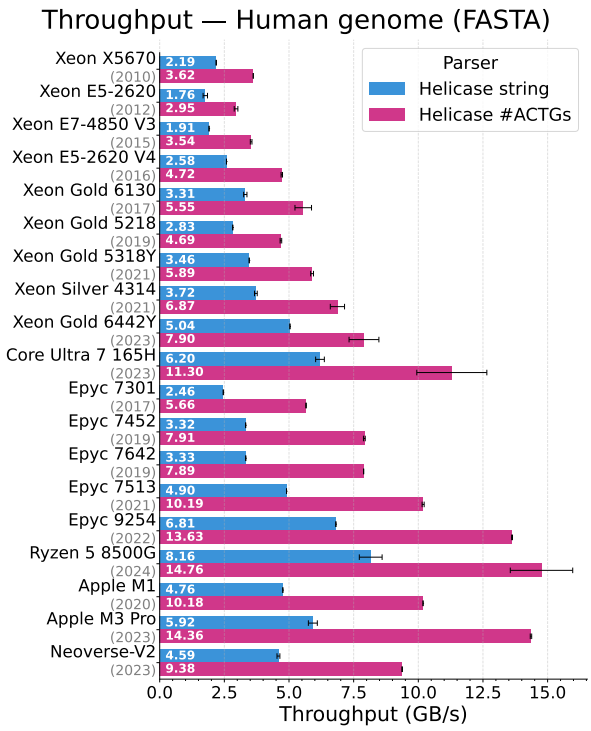
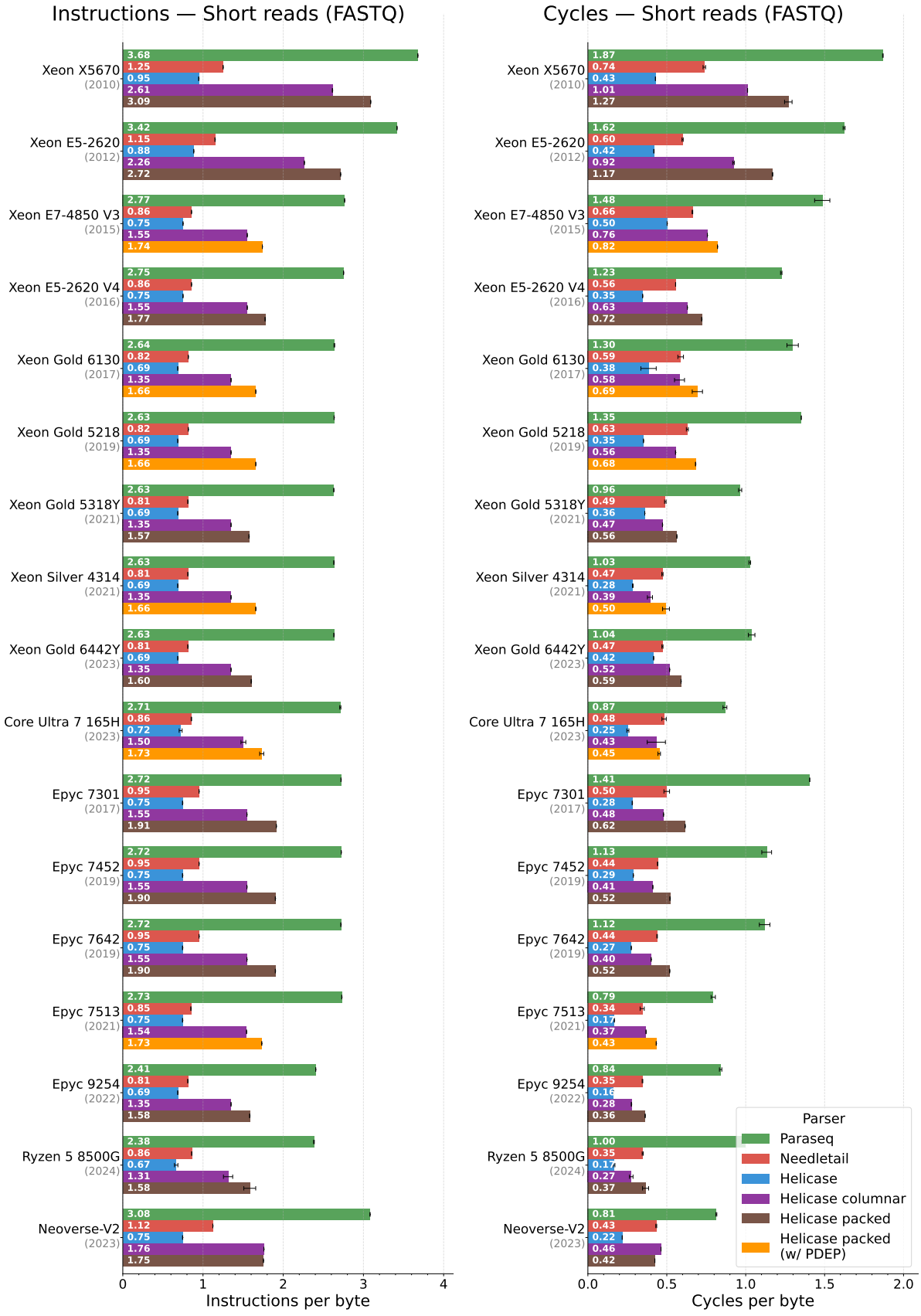


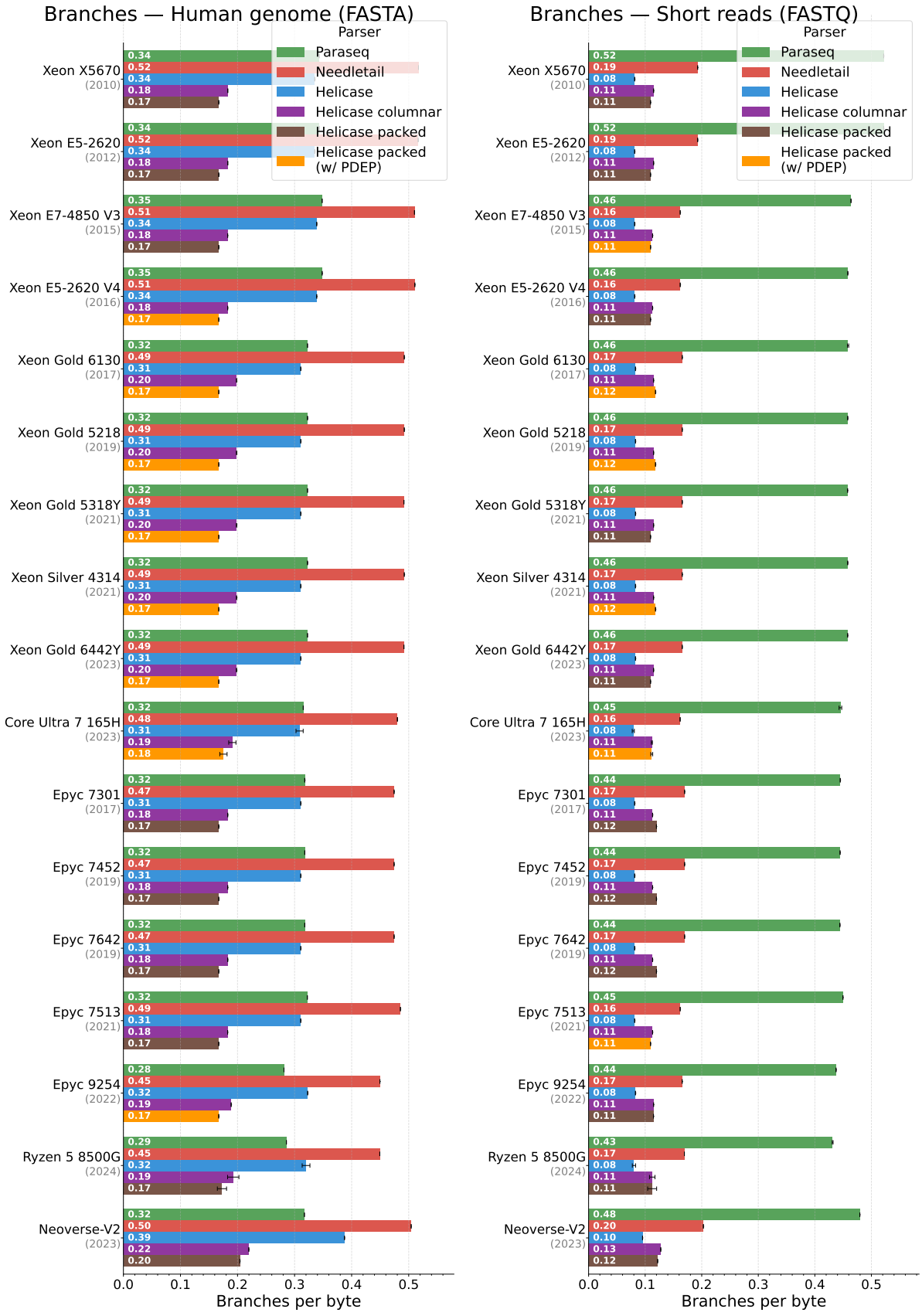
Figure B.3.: Throughput of Helicase string collection compared to counting DNA bases.



(a) Instructions per byte on short reads.

(b) Cycles per byte on short reads.

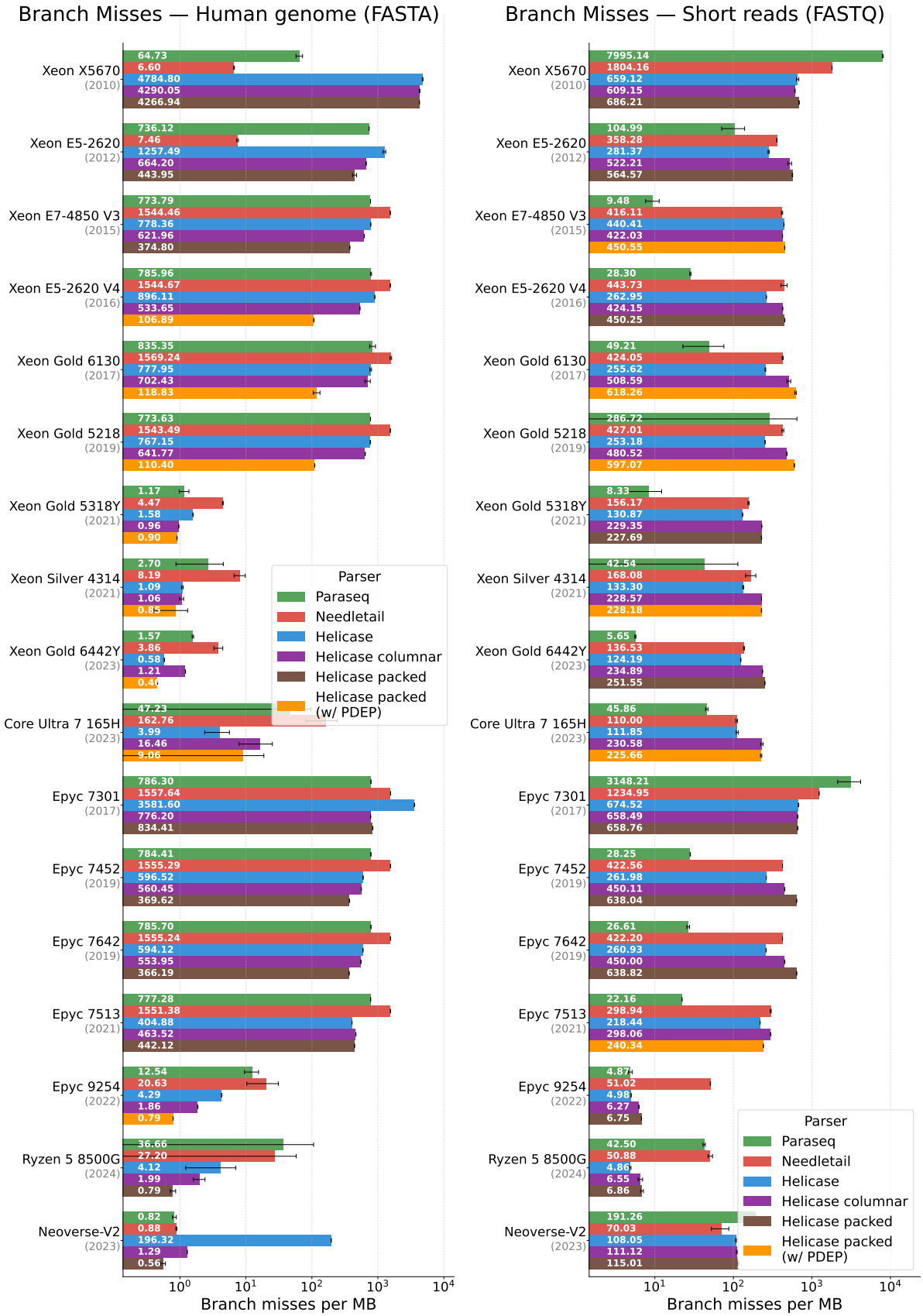
Figure B.4.: Instructions and cycles per byte on multiple CPUs, sorted by manufacturer and year.



(a) On a human genome (FASTA format).

(b) On short reads (FASTQ format).

Figure B.5.: Branches per byte on multiple CPUs, sorted by manufacturer and year.



(a) On a human genome (FASTA format).

(b) On short reads (FASTQ format).

Figure B.6.: Branch misses per MB on multiple CPUs, sorted by manufacturer and year.

C. Experiments on hyper-*k*-mers

C.1. Datasets

Table C.1 lists the datasets used in the experiments and their characteristics.

Table C.1.: E. Coli datasets used for benchmarking KFC. Min., Avg. and Max. refer to the minimum, average and maximum read length respectively.

Name	Type	Coverage	# reads	Total length	Min.	Avg.	Max.
SRR11434954	HiFi	5000×	1,789,131	23,122,913,014	46	12,924.1	26,294
SRR28370642	ONT Duplex	50×	114,703	236,908,842	349	2,065.4	126,029
SRR28370651	ONT Duplex	50×	109,061	226,502,819	330	2,076.8	125,012
SRR28370668	ONT Simplex	2000×	6,819,683	9,101,103,830	1	1,334.5	396,011

C.2. Multi-threading efficiency of KFC

In this section, we assess the multi-threading efficiency of KFC (Figure C.1). Our results demonstrate that KFC effectively utilizes multi-core architectures, achieving performance gains with up to several dozen cores before experiencing diminishing returns.

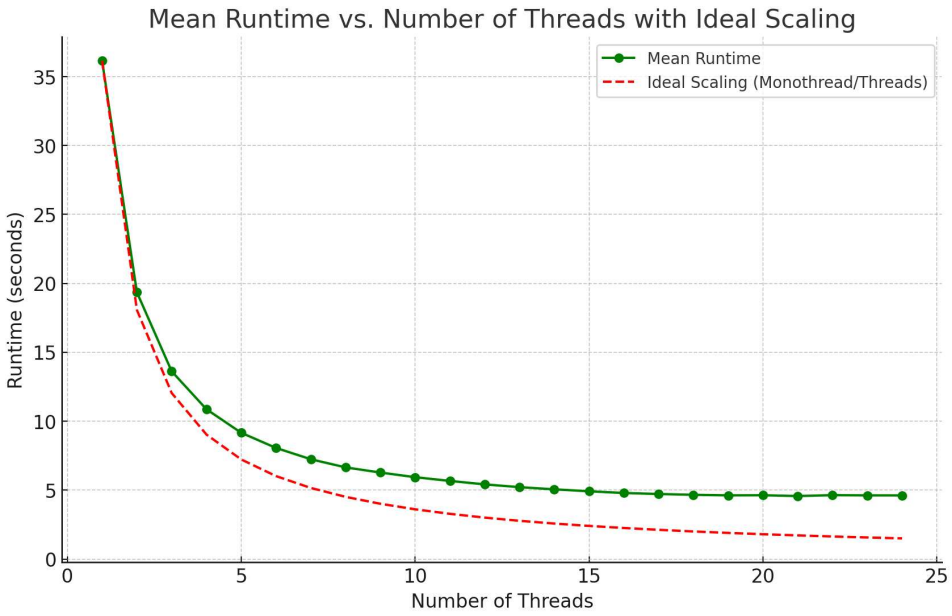
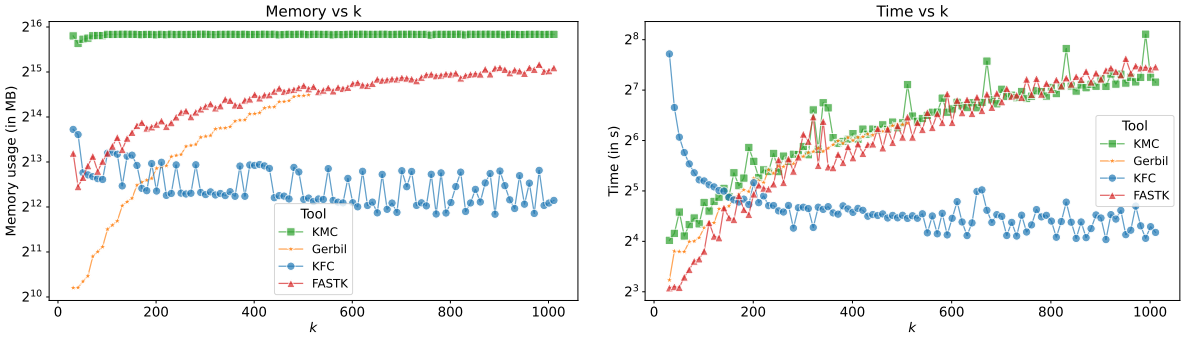


Figure C.1.: Multi-core usage efficiency of KFC on the 100× HiFi E. coli dataset.

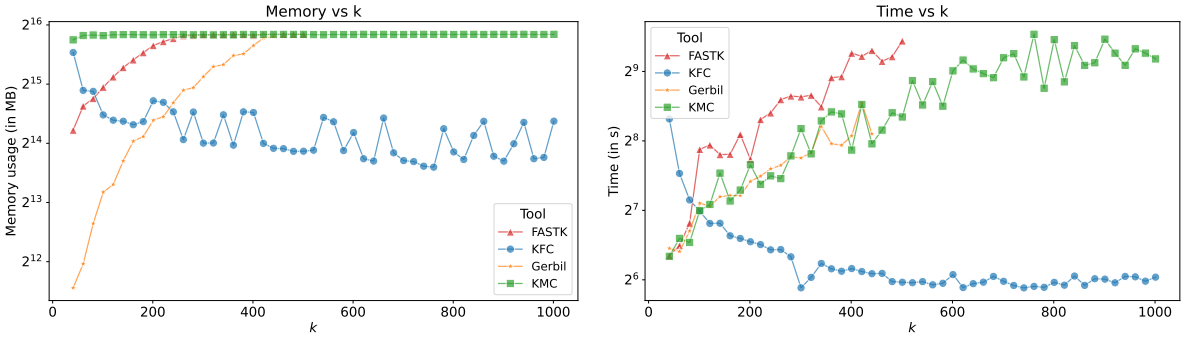
C.3. Metagenomic benchmarks

Figure C.2 and Figure C.3 confirm the trends of Section 14.4.1 on two larger HiFi metagenomic datasets.



(a) Memory usage (b) Running time

Figure C.2.: k-mer counting benchmark on HiFi Zymo community dataset (SRR13128014) downsampled at 5 gigabases, with unique k-mer filtering.



(a) Memory usage (b) Running time

Figure C.3.: k-mer counting benchmark on HiFi human gut datasets (SRR15275210, SRR15275211, SRR15275212, SRR15275213) downsampled at 15 gigabases, with unique k-mer filtering.

C.4. Effect of coverage

Figure C.4 shows benchmarks similar to Figure 14.7 but at various coverage levels, without significant change in the relative behaviors of the tools.

C.5. Effect of not filtering unique *k*-mers

In this section, we present the performance of KFC when unique k-mers are not filtered out. Figure C.6 shows the results discussed in Section 14.4.1. They are similar to Figure 14.7, but without filtering unique k-mers.

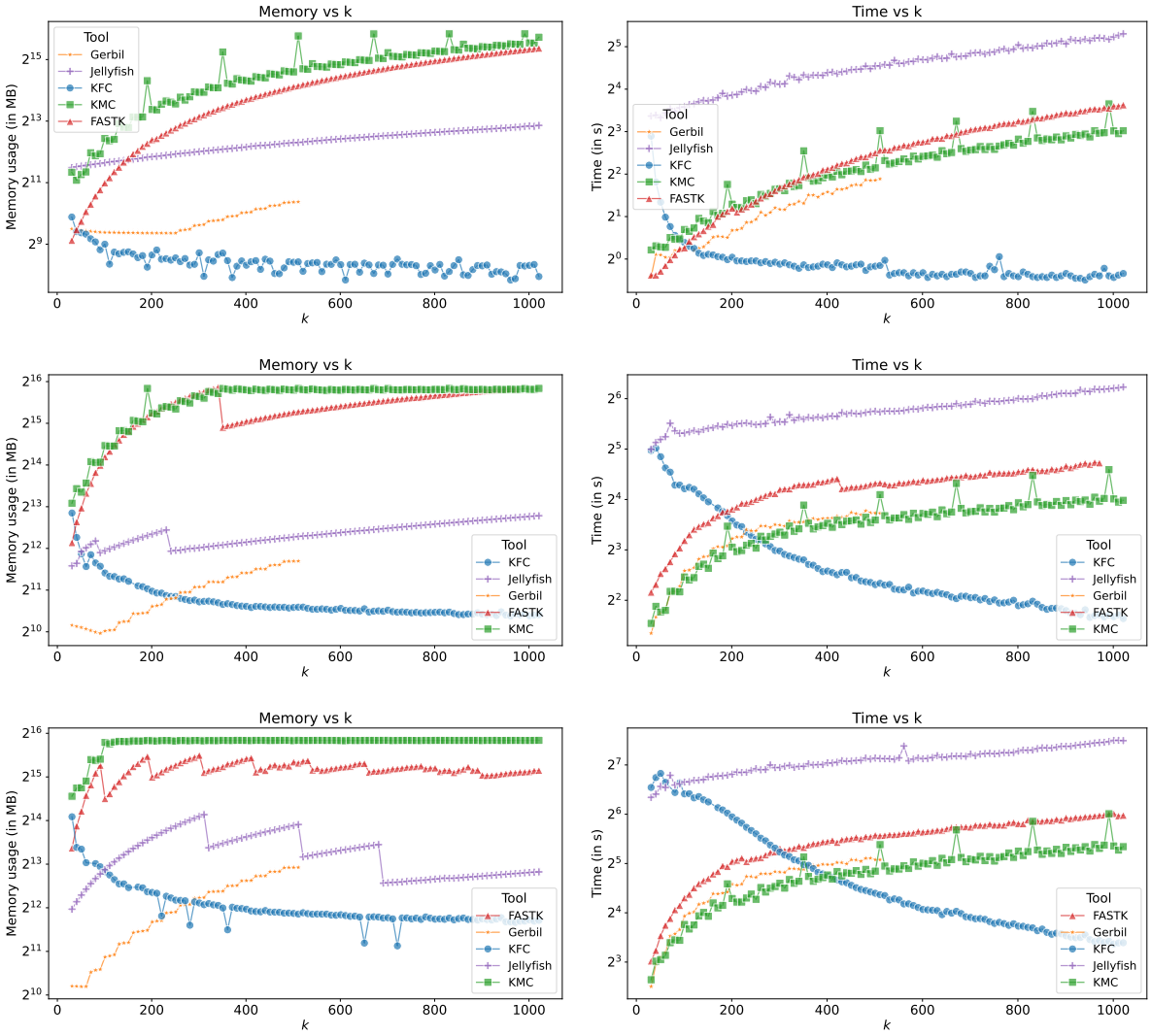


Figure C.4.: k-mer counting benchmark on ONT Simplex E. coli dataset (SRR28370668) for various coverage.

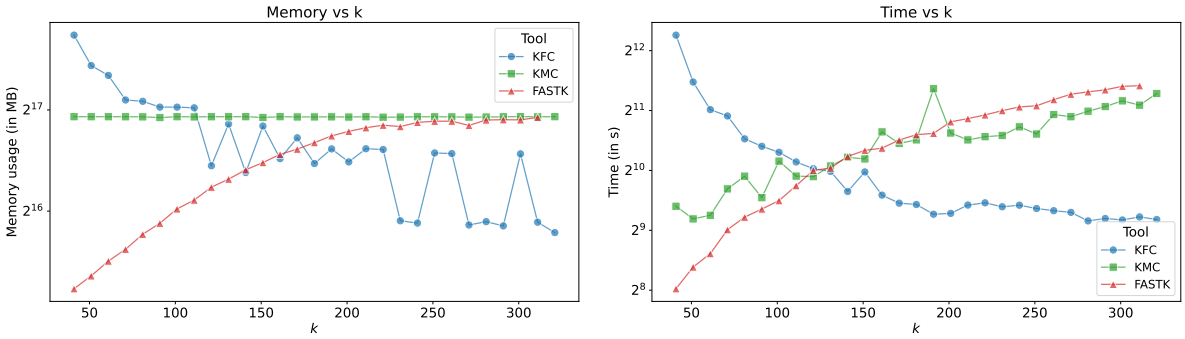


Figure C.5.: k-mer counting benchmark on the complete HiFi human gut datasets (SRR15275210, SRR15275211, SRR15275212, SRR15275213), filtering k-mers appearing once or twice (abundance threshold $t = 3$).

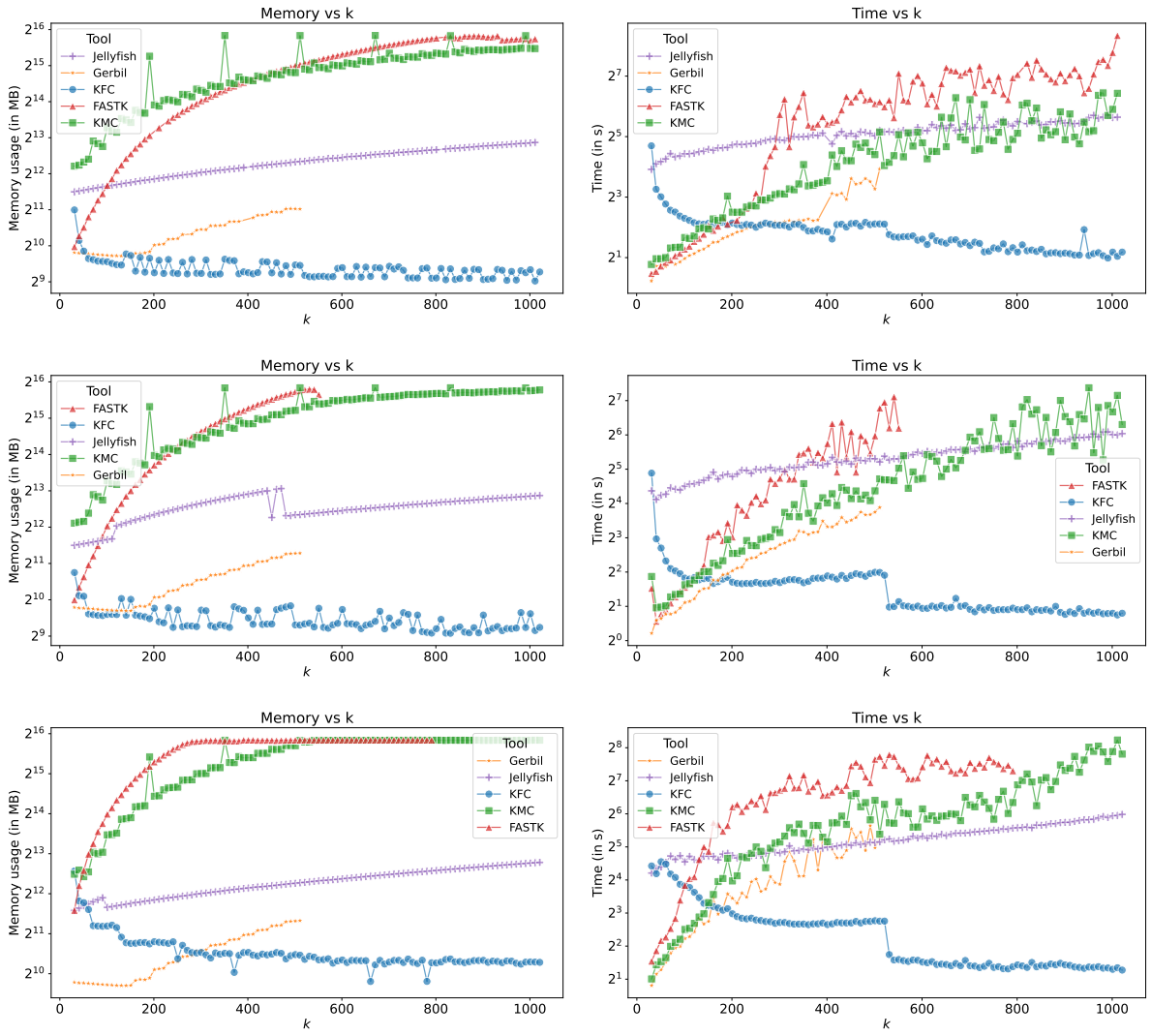
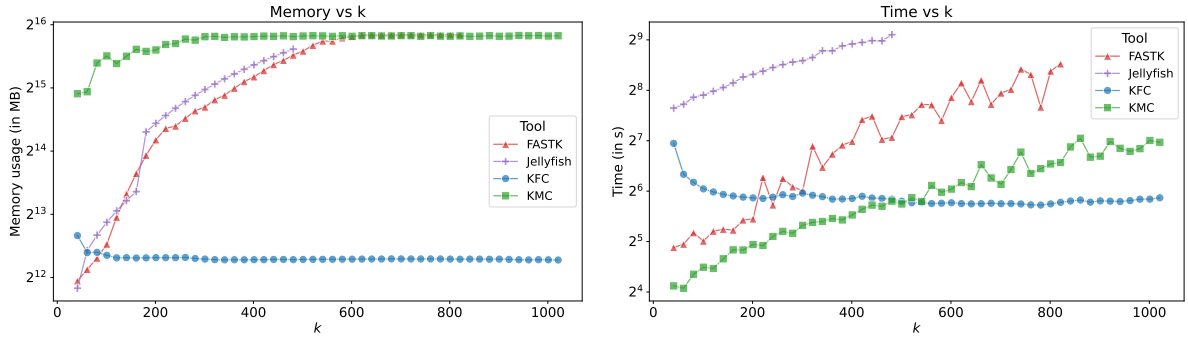


Figure C.6.: Comparison of k-mer benchmarks on different E. coli datasets without any filtering. Each subfigure shows the memory usage and timing plots for different sequencing technologies.

C.6. Pangenome benchmark

We evaluate the tools on one thousand *S. enterica* complete genomes from NCBI to assess the cost of counting large k-mers across a pangenome (Figure C.7). KFC remains the most memory-efficient tool ahead of FastK and KMC, and for very large k-mer sizes ($k > 500$) it also becomes the fastest, although its multithreading efficiency (analyzed in Section C.2) lags behind KMC on small input sizes, so it leads only when running on fewer threads.



(a) Memory usage

(b) Running time

Figure C.7.: k-mer counting benchmark on one thousand *S. enterica* complete genomes.

D. Proofs and experiments on multiminimizers

D.1. Proof of Theorem 16.1

Proof. We start by rewriting the density formula as

$$d = \lim_{M\to\infty} \frac{1}{\sigma^M} \sum_{S\in\Sigma^M} \frac{|\mathcal{S}_f(S)|}{|S|-k+1} = \lim_{M\to\infty} \frac{1}{M-k+1} \sum_{S\in\Sigma^M} \frac{1}{\sigma^M} \cdot |\mathcal{S}_f(S)| = \lim_{M\to\infty} \frac{\mathbb{E}[|\mathcal{S}_f(S)|]}{M-k+1} \tag{D.1}$$

where $\mathbb{E}[|\mathcal{S}_f(S)|]$ designates the expected particular density of a (uniform) random sequence S of length M, seen as a random variable. Suppose we can extend the original sequence S, of size M, and continue selecting minimizers past the first M characters. We define $\tau_M = \inf\{n \in \mathbb{N} : P_n^* > M\}$ as the random variable that determines how many minimizers have been selected in S; hence $\tau_M = |\mathcal{S}_f(S)|$. Therefore, we can rewrite the above as $d = \lim_{M\to\infty} \frac{\mathbb{E}[\tau_M]}{M-k+1}$.

Let us call $S_n = \sum_{i=1}^{n} \Delta_i$. Notice that, by definition of the Δ_i's, $S_n = P_n^*$; then $\tau_M = \inf\{n \in \mathbb{N} : S_n > M\}$ is a stopping time with respect to the σ-algebra generated by the Δ_i's. Using Wald's equation [Wal44], we get

$$\mathbb{E}[S_{\tau_M}] = \mathbb{E}\left[\sum_{i=1}^{\tau_M} \mathbb{E}[\Delta_i]\right] = \mathbb{E}\left[\tau_M \cdot \mu + \sum_{i=1}^{\tau_M} \varepsilon_i\right] = \mathbb{E}[\tau_M] \cdot \mu + \mathbb{E}\left[\sum_{i=1}^{\tau_M} \varepsilon_i\right].$$

We apply again Wald's equation to the right-hand term and obtain

$$\mathbb{E}[S_{\tau_M}] = \mathbb{E}[\tau_M] \cdot \mu + \mathbb{E}\left[\sum_{i=1}^{\tau_M} \mathbb{E}[\varepsilon_i]\right] = \mathbb{E}[\tau_M] \cdot \mu + \underbrace{\mathbb{E}[\tau_M] \cdot \mathbb{E}[\varepsilon]}_{=0}.$$

By definition of S_n, τ_M and the Δ_i's, notice that $M \le S_{\tau_M} \le M + w - 1$. Therefore

$$\frac{M}{M-k+1} \le \frac{\mathbb{E}[\tau_M]}{M-k+1} \cdot \mu \le \frac{M+w-1}{M-k+1},$$

and, taking the limit as $M \to \infty$, we establish our result. □

D.2. Proof of Proposition 16.1

Proof. We begin with the following lemma.

Lemma D.1 (Trapezoid rule)**.** For any function $f : [0,1] \to \mathbb{R}$ at least two times continuously derivable, we have

$$\frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m-1} f\Big(\frac{r}{\sigma^m}\Big) = \int_0^1 f(x)\,dx - \frac{f(1)+f(0)}{2\sigma^m} + O\Big(\frac{1}{\sigma^{2m}}\Big).$$

Proof. This is a direct application of the trapezoid rule for numerical integration. Defining $E = \int_0^1 f(x)\,dx - \frac{1}{\sigma^m}\Big(\frac{f(1)+f(0)}{2} + \sum_{r=1}^{\sigma^m-1} f(\frac{r}{\sigma^m})\Big)$, we have $E = -\frac{1}{12\sigma^{2m}} f''(\eta)$ for some $\eta \in [0,1]$ [Atk08, eq. (5.1.17)]. □

Let $1 \leq i \leq w$. For R_i to be the minimizer of the window $R_1, \dots, R_w$, then we must have:

$$\overbrace{R_1, \dots, R_{i-1}}^{>R_i}, \boxed{R_i}, \overbrace{R_{i+1}, \dots, R_w}^{\geq R_i}$$

The events $(R_j > R_i)$ and $(R_j \geq R_i)$ are clearly not independent of R_i; but resorting to the law of total probability, and using that the R_j's are i.i.d., we obtain

$$\begin{aligned}
\mathbb{P}(P_1 = i) &= \sum_{r=1}^{\sigma^m} \mathbb{P}(R_1 > r)^{i-1} \cdot \mathbb{P}(R_1 \geq r)^{w-i} \cdot \mathbb{P}(R_i = r) \\
&= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left(\frac{\sigma^m - r}{\sigma^m}\right)^{i-1} \cdot \left(\frac{\sigma^m - r}{\sigma^m} + \frac{1}{\sigma^m}\right)^{w-i} \\
&= \frac{1}{\sigma^m} \sum_{r=0}^{\sigma^m - 1} \left(\frac{r}{\sigma^m}\right)^{i-1} \cdot \left(\frac{r}{\sigma^m} + \frac{1}{\sigma^m}\right)^{w-i} \quad \text{by change of variable.}
\end{aligned}$$

Using Lemma D.1 with $f : x \mapsto x^{i-1}(x + \sigma^{-m})^{w-i}$, we obtain

$$\mathbb{P}(P_1 = i) = \frac{\mathbf{1}_{i=1}}{\sigma^{mw}} + \int_0^1 x^{i-1}(x + \sigma^{-m})^{w-i}\, dx - \frac{(1 + \sigma^{-m})^{w-i} + \sigma^{-mw}\mathbf{1}_{i=1}}{2\sigma^m} + O\left(\frac{1}{\sigma^{2m}}\right).$$

Using the fact that $(x + \sigma^{-m})^n = x^n + \frac{nx^{n-1}}{\sigma^m} + O(\sigma^{-2m})$, we get

$$\begin{aligned}
\mathbb{P}(P_1 = i) &= \int_0^1 x^{w-1}\, dx + \frac{w-i}{\sigma^m} \int_0^1 x^{w-2}\, dx - \frac{1}{2\sigma^m} + O\left(\frac{1}{\sigma^{2m}}\right) \\
&= \frac{1}{w} + \left(\frac{w-i}{w-1} - \frac{1}{2}\right) \cdot \frac{1}{\sigma^m} + O\left(\frac{1}{\sigma^{2m}}\right).
\end{aligned}$$

□

D.3. Convergence of P_1 to the uniform distribution

In Figure D.1, we computed the distribution of P_1 for various values of m, using the exact formula

$$\mathbb{P}(P_1 = i) = \frac{1}{\sigma^m} \sum_{r=0}^{\sigma^m - 1} \left(\frac{r}{\sigma^m}\right)^{i-1} \cdot \left(\frac{r+1}{\sigma^m}\right)^{w-i}$$

rather than the approximation of Proposition 16.1. As one can see, P_1 converges quickly to the limit uniform distribution (as one can expect since the dependency on m is of the order of σ^{-m}).

D.4. Monte-Carlo simulations of random minimizers

Simulating our probabilistic model of random minimizers, with Hypothesis 16.2, we obtain the results depicted in Figure D.2, which have been commented on in the core of the paper.

One thing that seems surprising at first glance is the peak observed for $\mathbb{E}[\Delta_2]$. When sliding the window by one position, either the previous minimizer X is still in the window and we change the minimizer only if the new value X' is $< X$, or X is not in the window anymore and we select as a new minimizer the minimum value in the window (which is called a *rescan*). Here, P_1 is always obtained by a rescan operation, meaning that P_2^* is always obtained as "the next minimizer following a rescan". We do not observe any other peak in the distribution of $\mathbb{E}[\Delta_i]$'s as the other rescans are randomly found in the sequences, and therefore their impact is smoothed among all simulations. We have the following result; note that both the formula and the empirical simulations yield a value of $\mathbb{E}[\Delta_2] \approx 5.87$ for $w = 10$ and $m = 8$.

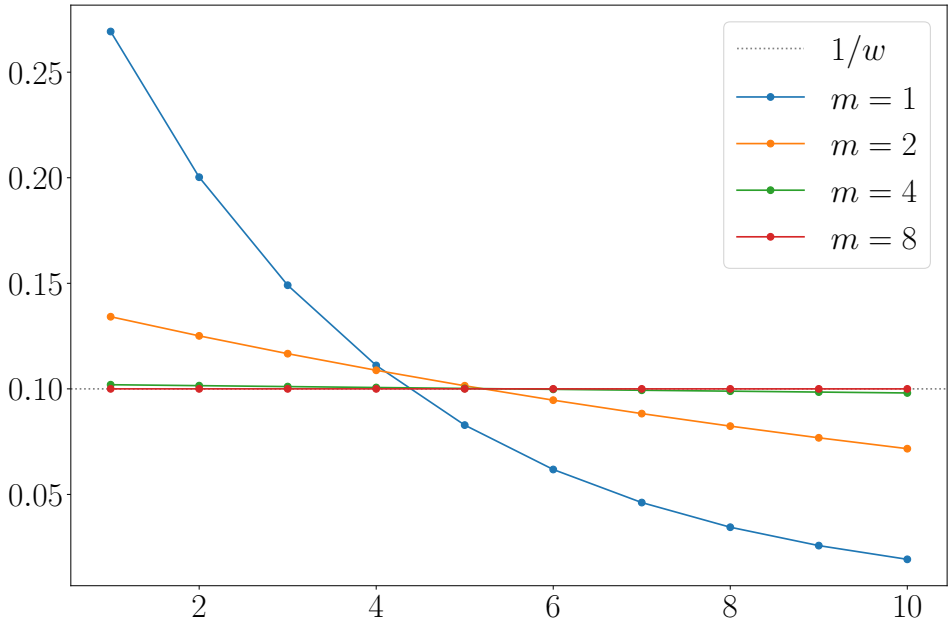
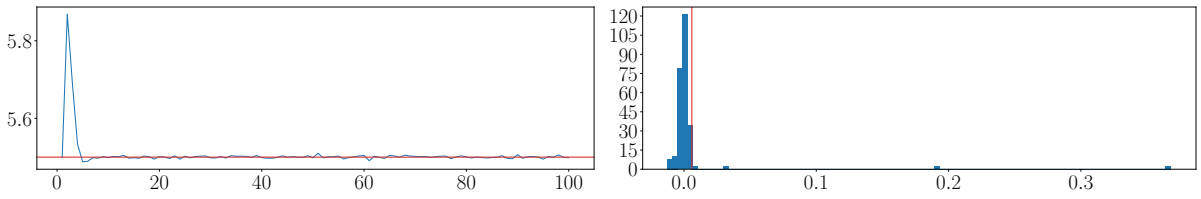


Figure D.1.: With $w = 10$, evolution of the distribution of P_1 as m increases. The actual value of P_1 is computed using the exact formula instead of the approximation of Proposition 16.1.



(a) Empirical values of $\mathbb{E}[\Delta_i]$ for $1 \leq i \leq 100$. The red line is equal to $\mu = \frac{w+1}{2}$.

(b) Empirical distribution of ε. The red line denotes the empirical mean of ε.

Figure D.2.: Monte-Carlo simulations of our model of random minimizers under Hypothesis 16.2. We used $m = 8$, $w = 10$ and generated $N_{\text{simu}} = 10^6$ random sequences $\mathbf{R}$, all long enough so that the sequence Δ contains at least 100 values.

Proposition D.1 (Expected value of Δ_2 (simplified))**.** Assuming $w^2 = o(\sigma^m)$,

$$\mathbb{E}[\Delta_2] = \frac{3\ln 2 - 1}{2} \cdot w + \frac{\ln 2}{2} + \frac{1}{8} - \frac{w-1}{32w^2} + O\Big(\frac{1}{\sigma^m}\Big).$$

We actually prove the following, more general, result:

Proposition D.2 (Expected value of Δ_2 (general))**.**

$$\mathbb{E}[\Delta_2] = (H_{2w} - H_w)\frac{3w+1}{2} - \frac{w-1}{2} + O\Big(\frac{1}{\sigma^m}\Big)$$

where H_n denotes the n-th harmonic number $1 + \frac{1}{2} + \cdots + \frac{1}{n}$.

Proof. To prove this, we distinguish two cases: either P_2^* "overwrites" P_1^* because $R_{P_2^*} < R_{P_1^*}$, or P_1^* goes out of scope and P_2^* is selected by a rescan. Therefore,

$$\mathbb{E}[\Delta_2] = \mathbb{E}[\Delta_2 \cdot \mathbf{1}_{P_2^* \text{ overwrites } P_1^*}] + \mathbb{E}[\Delta_2 \cdot \mathbf{1}_{\text{rescan}}].$$

In the following paragraphs, we show that:

- $\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{P_2^* \text{ overwrites } P_1^*}] = w(H_{2w} - H_w) - \frac{w-1}{2} + O(1/\sigma^m)$
- $\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{\text{rescan}}] = \frac{w+1}{2}(H_{2w} - H_w) + O(1/\sigma^m)$

Combining these,

$$\mathbb{E}[\Delta_2] = (H_{2w} - H_w)\frac{3w+1}{2} - \frac{w-1}{2} + O\Big(\frac{1}{\sigma^m}\Big).$$

□

By approximating H_n as $\ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O\big(\frac{1}{n^4}\big)$, we get the desired result:

$$\mathbb{E}[\Delta_2] = \frac{3\ln 2 - 1}{2} \cdot w + \frac{\ln 2}{2} + \frac{1}{8} - \frac{w-1}{32w^2} + O\Big(\frac{1}{\sigma^m}\Big).$$

Overwrite case.

Lemma D.2. For any $1 \le j \le i \le w$, the probability that $P_2^* = w + j$ overwrites $P_1^* = i$ is

$$\mathbb{P}((P_1^* = i) \cap (P_2^* = w+j) \cap (R_{w+j} < R_i)) = \frac{1}{w+j-1} - \frac{1}{w+j} + O\Big(\frac{1}{\sigma^m}\Big).$$

Proof. Let $O_{i,w+j}$ denote the event $(P_1^* = i) \cap (P_2^* = w+j) \cap (R_{w+j} < R_i)$.

$$\begin{aligned}
\mathbb{P}(O_{i,w+j}) &= \sum_{r=1}^{\sigma^m} \mathbb{P}(R_1 > r)^{i-1} \cdot \mathbb{P}(R_i = r) \cdot \mathbb{P}(R_1 \ge r)^{w-i+j-1} \cdot \mathbb{P}(R_{w+j} < r) \\
&= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \mathbb{P}(R_1 > r)^{i-1} \cdot \mathbb{P}(R_1 \ge r)^{w-i+j-1} \cdot (1 - \mathbb{P}(R_{w+j} \ge r)) \\
&= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left(\frac{\sigma^m - r}{\sigma^m}\right)^{i-1} \left(\frac{\sigma^m - r + 1}{\sigma^m}\right)^{w-i+j-1} \left(1 - \frac{\sigma^m - r + 1}{\sigma^m}\right) \\
&= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left(\frac{r-1}{\sigma^m}\right)^{i-1} \left(\frac{r}{\sigma^m}\right)^{w-i+j-1} \left(1 - \frac{r}{\sigma^m}\right) \quad \text{by change of variable.}
\end{aligned}$$

Applying Lemma D.1 with $f : x \mapsto (x-\sigma^{-m})^{i-1}x^{w-i+j-1}(1-x)$, we obtain

$$\mathbb{P}(O_{i,w+j}) = \int_0^1 (x-\sigma^{-m})^{i-1}x^{w-i+j-1}(1-x)\,dx - \frac{\mathbf{1}_{w+j=i+1}(-\sigma^{-m})^{i-1}}{2\sigma^m} + O\Big(\frac{1}{\sigma^{2m}}\Big).$$

Using the expansion $(x-\sigma^{-m})^{i-1} = x^{i-1} + O(\sigma^{-m})$, we get

$$\begin{aligned}\mathbb{P}(O_{i,w+j}) &= \int_0^1 x^{w+j-2}(1-x)\,dx + O\Big(\frac{1}{\sigma^m}\Big)\\ &= \frac{1}{w+j-1} - \frac{1}{w+j} + O\Big(\frac{1}{\sigma^m}\Big),\end{aligned}$$

since $\int_0^1 x^n(1-x)\,dx = \frac{1}{n+1} - \frac{1}{n+2}$. □

Assuming $w^2 = o(\sigma^m)$,

$$\begin{aligned}\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{P_2^* \text{ overwrites } P_1^*}] &= \sum_{i=1}^{w}\sum_{j=1}^{i}(w-i+j)\cdot\mathbb{P}(O_{i,w+j})\\ &= \sum_{i=1}^{w}\sum_{j=1}^{i}(w-i+j)\left[\frac{1}{w+j-1} - \frac{1}{w+j} + O\Big(\frac{1}{\sigma^m}\Big)\right]\\ &= \sum_{i=1}^{w}\left[\sum_{j=1}^{i}\frac{w-i+j}{w+j-1} - \sum_{j=1}^{i}\frac{w-i+j}{w+j}\right] + O\Big(\frac{1}{\sigma^m}\Big)\\ &= \sum_{i=1}^{w}\left[\sum_{j=1}^{i}\frac{w-i+j-1+1}{w+j-1} - \left(i - i\sum_{j=1}^{i}\frac{1}{w+j}\right)\right] + O\Big(\frac{1}{\sigma^m}\Big)\\ &= \sum_{i=1}^{w}\left[i - (i-1)\sum_{j=1}^{i}\frac{1}{w+j-1} - i + i\sum_{j=1}^{i}\frac{1}{w+j}\right] + O\Big(\frac{1}{\sigma^m}\Big)\\ &= \sum_{i=1}^{w}\left[\sum_{j=1}^{i}\frac{1}{w+j-1} + i\left(\frac{1}{w+i} - \frac{1}{w}\right)\right] + O\Big(\frac{1}{\sigma^m}\Big).\end{aligned}$$

Expanding into three different sums,

$$\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{P_2^* \text{ overwrites } P_1^*}] = \underbrace{\sum_{i=1}^{w}\sum_{j=1}^{i}\frac{1}{w+j-1}}_{(A)} + \underbrace{\sum_{i=1}^{w}\frac{i}{w+i}}_{(B)} - \underbrace{\frac{1}{w}\sum_{i=1}^{w} i}_{(C)} + O\Big(\frac{1}{\sigma^m}\Big).$$

(C) is obviously $\frac{w+1}{2}$. For (A), note that the term $\frac{1}{w}$ appears w times, $\frac{1}{w+1}$ appears $w-1$ times, and so on up to $\frac{1}{2w-1}$ which appears only once. Hence,

$$(A) = \sum_{i=0}^{w-1}\frac{w-i}{w+i} = 1 + \sum_{i=1}^{w}\frac{w-i}{w+i} = 1 + w(H_{2w} - H_w) - \underbrace{\sum_{i=1}^{w}\frac{i}{w+i}}_{(B)}.$$

We do not need to compute (B) as terms cancel, and

$$\begin{aligned}\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{P_2^* \text{ overwrites } P_1^*}] &= 1 + w(H_{2w} - H_w) - (B) + (B) - \frac{w+1}{2} + O\Big(\frac{1}{\sigma^m}\Big)\\ &= w(H_{2w} - H_w) - \frac{w-1}{2} + O\Big(\frac{1}{\sigma^m}\Big).\end{aligned}$$

Rescan case.

Let $1 \le r \le \sigma^m$ and $\alpha(r) = \frac{\sigma^m - r + 1}{\sigma^m}$. We use the truncated trapezoid approximation:

$$\frac{1}{\sigma^m} \sum_{s=1}^{\sigma^m - r} f\Big(\frac{s}{\sigma^m}\Big) = \int_0^{\alpha(r)} f(x)\,dx + O\Big(\frac{1}{\sigma^m}\Big)$$

for any twice continuously derivable f.

Lemma D.3. For any $1 \le i, j \le w$, the probability that $P_2^* = i + j$ as a rescan when $P_1^* = i$ is

$$\mathbb{P}((P_1^* = i) \cap (P_2^* = i + j) \cap (\text{rescan})) = \frac{1}{w(w+i)} + O\Big(\frac{1}{\sigma^m}\Big).$$

Proof. Let us denote by $R_{i,j}$ the event "$(P_1^* = i) \cap (P_2^* = i + j) \cap (\text{rescan})$" for any $1 \le i, j \le w$. When this event is realized, then all variables

- $R_1, \dots, R_{i-1}$ are $> R_i$;
- $R_{i+1}, \dots, R_{w+i}$ are $\ge R_i$ (including R_{i+j});
- $R_{i+1}, \dots, R_{i+j-1}$ are $> R_j$;
- $R_{i+j+1}, \dots, R_{w+i}$ are $\ge R_j$.

Therefore,

$$\begin{aligned}
\mathbb{P}(R_{i,j}) &= \sum_{r=1}^{\sigma^m} \sum_{s=1}^{\sigma^m} \mathbb{P}(R_1 > r)^{i-1} \cdot \mathbb{P}(R_i = r) \cdot \mathbb{P}((R_1 \ge r) \cap (R_1 > s))^{j-1} \\
&\qquad \cdot \mathbb{P}((R_{i+j} = s) \cap (R_{i+j} \ge r)) \cdot \mathbb{P}((R_1 \ge r) \cap (R_1 \ge s))^{w-j} \\
&= \frac{1}{\sigma^{2m}} \sum_{r=1}^{\sigma^m} \mathbb{P}(R_1 > r)^{i-1} \sum_{s=r}^{\sigma^m} \mathbb{P}(R_1 > s)^{j-1} \cdot \mathbb{P}(R_1 \ge s)^{w-j} \\
&= \frac{1}{\sigma^{2m}} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \sum_{s=r}^{\sigma^m} \Big(\frac{\sigma^m - s}{\sigma^m}\Big)^{j-1} \Big(\frac{\sigma^m - s + 1}{\sigma^m}\Big)^{w-j} \\
&= \frac{1}{\sigma^{2m}} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \sum_{s=0}^{\sigma^m - r} \Big(\frac{s}{\sigma^m}\Big)^{j-1} \Big(\frac{s+1}{\sigma^m}\Big)^{w-j}.
\end{aligned}$$

Notice that the term $s = 0$ in the inner sum equals $\mathbf{1}_{j=1}\sigma^{-m(w-1)}$ which is $O(\sigma^{-m})$ as we suppose that $w > 1$. Applying the truncated trapezoid approximation to the function $f : x \mapsto x^{j-1}(x + \sigma^{-m})^{w-j}$, we get

$$\begin{aligned}
\mathbb{P}(R_{i,j}) &= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \left[\int_0^{\alpha(r)} f(x)\,dx + O\Big(\frac{1}{\sigma^m}\Big)\right] \\
&= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \int_0^{\alpha(r)} f(x)\,dx + O\Big(\frac{1}{\sigma^m}\Big).
\end{aligned}$$

Using the expansion $f(x) = x^{w-1} + O(\sigma^{-m})$, we get

$$\begin{aligned}
\mathbb{P}(R_{i,j}) &= \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \int_0^{\alpha(r)} x^{w-1}\,dx + O\Big(\frac{1}{\sigma^m}\Big) \\
&= \frac{1}{w} \cdot \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \Big(\frac{\sigma^m - r}{\sigma^m}\Big)^{i-1} \cdot \Big(\frac{\sigma^m - r + 1}{\sigma^m}\Big)^{w} + O\Big(\frac{1}{\sigma^m}\Big) \\
&= \frac{1}{w} \cdot \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \Big(\frac{r-1}{\sigma^m}\Big)^{i-1} \cdot \Big(\frac{r}{\sigma^m}\Big)^{w} + O\Big(\frac{1}{\sigma^m}\Big).
\end{aligned}$$

Applying Lemma D.1 with the function $f : x \mapsto (x - \sigma^{-m})^{i-1} x^w$, we get

$$\mathbb{P}(R_{i,j}) = \frac{1}{w} \int_0^1 (x - \sigma^{-m})^{i-1} x^w \, dx + O\left(\frac{1}{\sigma^m}\right) = \frac{1}{w(w+i)} + O\left(\frac{1}{\sigma^m}\right).$$

□

Finally, still assuming $w^2 = o(\sigma^m)$:

$$\mathbb{E}[\Delta_2 \cdot \mathbf{1}_{\text{rescan}}] = \sum_{i=1}^{w} \sum_{j=1}^{w} \frac{j}{w(w+i)} + O\left(\frac{1}{\sigma^m}\right) = \frac{w+1}{2}(H_{2w} - H_w) + O\left(\frac{1}{\sigma^m}\right).$$

D.5. Proof of Proposition 16.2

Proof. We have, from Definition 16.2:

$$d^* = \frac{1}{2^{\sigma^k} - 1} \sum_{n=1}^{\sigma^k} \binom{\sigma^k}{n} \mathbb{E}_{|\mathcal{X}|=n}[d^*(\mathcal{X})].$$

To compute the required expectation, let $\mathcal{X} = \{X_1, \dots, X_n\}$ be a set of k-mers chosen uniformly at random. We denote by $M_i \in [\![1, \sigma^m]\!]$ the random variable denoting the minimizer of k-mer X_i (assimilating minimizers and their rank according to the chosen order $\mathcal{O}_m$). Under Hypothesis 16.2, it is not difficult to establish that $\mathbb{P}(M_i > r) = \mathbb{P}(X_1 > r)^w$ where X_i denotes the i-th m-mer of X_i and $w = k - m + 1$. Therefore, we obtain

$$\mathbb{P}(M_i = r) = \left(\frac{\sigma^m - r + 1}{\sigma^m}\right)^w - \left(\frac{\sigma^m - r}{\sigma^m}\right)^w.$$

We denote by P_r the number of k-mers in $\mathcal{X}$ that admit r as their minimizer; then:

$$\begin{aligned}
\mathbb{E}[|\operatorname{MinCover}(\mathcal{X})|] &= \sum_{r=1}^{\sigma^m} \mathbb{E}[\mathbf{1}_{P_r>0}] = \sum_{r=1}^{\sigma^m} (1 - \mathbb{P}(P_r = 0)) = \sigma^m - \sum_{r=1}^{\sigma^m} (1 - \mathbb{P}(M_1 = r))^n \\
&= \sigma^m \left[1 - \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left[1 - \left(\frac{\sigma^m - r + 1}{\sigma^m}\right)^w + \left(\frac{\sigma^m - r}{\sigma^m}\right)^w\right]^n\right] \\
&= \sigma^m \left[1 - \frac{1}{\sigma^m} \sum_{r=1}^{\sigma^m} \left[1 - \left(\frac{r}{\sigma^m}\right)^w + \left(\frac{r-1}{\sigma^m}\right)^w\right]^n\right].
\end{aligned}$$

This is exactly the standard coupon collector's problem, where the coupons are distinct minimizers and $|\mathcal{X}|$ is the number of coupons bought. Substituting and simplifying yields Proposition 16.2. □

D.6. Proof of Theorem 16.2

We prove the NP-completeness of MultiMinCover by (i) showing first that the problem is in NP and then (ii) by reduction from SetCover. An alternative reduction from VertexCover is presented at the end of this section.

D.6.1. MultiMinCover is in NP

For the decision version of MultiMinCover, given a proposed set $\mathcal{Y}$ of m-mers of cardinality $\leq k$, one can verify whether $\mathcal{Y}$ is a multiminimizer cover of $\mathcal{X}$ by iterating over k-mers in $\mathcal{X}$, computing their N minimizers, and checking whether at least one belongs to $\mathcal{Y}$. This can be done in $O(|\mathcal{X}| \cdot N)$, hence the result.

D.6.2. Reduction from SetCover

Let $\mathcal{U} = \{1, \dots, n\}$ and let $\mathcal{S} \subseteq 2^{\mathcal{U}}$ be a collection of subsets of $\mathcal{U}$, whose union is $\mathcal{U}$. We recall that the SetCover problem amounts to finding the smallest subcollection of $\mathcal{S}$ whose union is $\mathcal{U}$. The *frequency* of an element $u \in \mathcal{U}$ is defined as the number of subsets in $\mathcal{S}$ that contain u. We define $N = \max_{u \in \mathcal{U}} \text{freq}(u)$ as the maximum frequency of any element of $\mathcal{U}$. Note that $N \leq |\mathcal{S}|$.

To illustrate the proof, we work through an example. Let $\mathcal{U} = \{1, \dots, 5\}$ and $\mathcal{S} = \{\{1,2,3\}, \{4,5\}, \{2,4\}, \{3,4\}\}$. The optimal solution for this instance of set cover is $\{1,2,3\}, \{4,5\}$ of cardinality 2. The maximal frequency is $N = 3$ (since 4 appears in 3 subsets).

We use the following 3-letter alphabet $\Sigma = \{0, 1, a\}$. Let $m = \lceil \log_2 |\mathcal{S}| \rceil$. Take any indexing $S_1, \dots, S_{|\mathcal{S}|}$ of the subsets forming $\mathcal{S}$; we associate to each subset S_i the m-mer P_i corresponding to i in binary coded with m bits. Then, let $k = Nm + N - 1 = N(m+1) - 1$. To each element $u \in \mathcal{U}$, let $i_1, \dots, i_f$ be the indices of the S_i's containing u, with $f = \text{freq}(u)$. We associate to u the k-mer $X_u = m_{i_1} a m_{i_2} a \cdots a m_{i_f} a^{(N-f)(m+1)}$.

Example D.1. We have $m = \lceil \log_2 4 \rceil = 2$. We associate $m_1 = 00$ to $\{1,2,3\}$, $m_2 = 01$ to $\{4,5\}$, $m_3 = 10$ to $\{2,4\}$ and $m_4 = 11$ to $\{3,4\}$. We have $k = 8$ and the following k-mers:

$$X_1 = \underline{00}aaaaaa, \quad X_2 = \underline{00}a\underline{10}aaa, \quad X_3 = \underline{00}a\underline{11}aaa, \quad X_4 = \underline{01}a\underline{10}a\underline{11}, \quad X_5 = \underline{01}aaaaaa.$$

The goal now is to exhibit N orders $\mathcal{O}_m^1, \dots, \mathcal{O}_m^N$ on m-mers, so that the following is true: for each k-mer x, its N minimizers correspond exactly to its m-mers coming from the m_i's; i.e. its minimizers contain only the letters `0` and `1`. It should be clear that any solution to MultiMinCover in this context is equivalent to a solution of the original SetCover instance. Since there are $|\mathcal{U}| = n$ k-mers of size $N(m+1) - 1 = O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ and exactly $|\mathcal{S}|$ minimizers to choose from, the reduction is indeed polynomial, hence the result — once we prove said orders on m-mers can indeed be constructed in polynomial time.

By construction, we want the m-mers $m_1, \dots, m_{|\mathcal{S}|}$ to receive a rank (for all orders) between 1 and $|\mathcal{S}|$; all subsequent possible m-mers (either binary, or containing at least one `a`) should receive a rank $> |\mathcal{S}| + 1$, and their relative order is of no importance since they will not be selected as minimizers whatsoever. By doing so, we ensure that the N minimizers of each k-mer X_u can only be the m_i's it is made of. If X_u contains several such m_i's, the only thing left to do is to ensure that each of them is indeed a minimizer for at least one order – so that the number of distinct minimizers of X_u is equal to the frequency of u in the original SetCover instance — thus ensuring the 1-to-1 correspondence between solutions of the two problems.

We exhibit Algorithm DetermineOrders (Algorithm D.1) to construct those orders.

At this point, we have determined that any m-mer containing at least one `a` has rank at least 5 in any of the 3 orderings to build, and $\{00, 01, 10, 11\}$ must receive ranks 1 to 4. We initialize two arrays, `ranks` and `minimizers`, following Algorithm DetermineOrders. Applying Lines 7–9 to X_1 and X_5, we get:

`rank`	$\mathcal{O}_m^1$	$\mathcal{O}_m^2$	$\mathcal{O}_m^3$
00			
01			
10			
11			

`minimizers`	$\min_1(x)$	$\min_2(x)$	$\min_3(x)$	`cand`(x)
X_1	00	00	00	$\emptyset$

`minimizers`	$\min_1(x)$	$\min_2(x)$	$\min_3(x)$	`cand`(x)
X_2				00, 10
X_3				00, 11
X_4				01, 10, 11
X_5	01	01	01	$\emptyset$

Now, consider X_2 and the m-mer 00. Applying Lines 11–15, choosing $l = 1$, we get:

`rank`	$\mathcal{O}_m^1$	$\mathcal{O}_m^2$	$\mathcal{O}_m^3$
00	1		
01			
10			
11			

`minimizers`	$\min_1(x)$	$\min_2(x)$	$\min_3(x)$	`cand`(x)
X_1	00	00	00	$\emptyset$
X_2	00			10
X_3	00			11
X_4				01, 10, 11
X_5	01	01	01	$\emptyset$

For X_2 and X_3, their respective remaining candidate fills the empty spots in `minimizers` by applying again Lines 7–9. Considering X_4, a possible outcome could be the following:

`rank`	$\mathcal{O}_m^1$	$\mathcal{O}_m^2$	$\mathcal{O}_m^3$
00	1		
01	2		
10		1	
11			

`minimizers`	$\min_1(x)$	$\min_2(x)$	$\min_3(x)$	`cand`(x)
X_1	00	00	00	$\emptyset$
X_2	00	10	10	$\emptyset$
X_3	00	11	11	$\emptyset$
X_4	01	10	11	$\emptyset$
X_5	01	01	01	$\emptyset$

After which, any completion of `ranks` would work.

Proposition D.3. Algorithm DetermineOrders is correct and runs in polynomial time.

Proof. Each iteration in the `while` loop eliminates one candidate minimizer, of which there are at most $|\mathcal{U}| \cdot N$. For a given candidate minimizer, if Lines 7–9 are executed, the `for` loop takes at most N steps; and if Lines 11–15 are executed, parsing a column in `ranks` takes $|\mathcal{S}|$ steps, and the `for` loop at most $|\mathcal{U}|$ steps. Since $N \leq |\mathcal{S}|$, we end up with a worst-case complexity of $O(|\mathcal{U}| \cdot |\mathcal{S}| \cdot (|\mathcal{S}| + |\mathcal{U}|))$, polynomial as claimed.

Algorithm D.1: DetermineOrders algorithm.

function DETERMINEORDERS($m_1, \dots, m_{|\mathcal{S}|}$; $X_1, \dots, X_n$; N)
  Initialize `ranks`, a $|\mathcal{S}| \times N$ array
  Initialize `minimizers`, a $n \times N$ array
  for $1 \leq i \leq n$ **do**
    Let $\texttt{cand}(X_i) \subseteq \{m_1, \dots, m_{|\mathcal{S}|}\}$ be the set of candidate minimizers of X_i
  while $\exists\, i$ such that $\texttt{cand}(X_i) \neq \emptyset$ **do**
    Let $m_j \in \texttt{cand}(X_i)$
    if $\texttt{cand}(X_i) \setminus \{m_j\} = \emptyset$ **then** *// The relative rank of m_j does not matter.*
      for all l such that $\texttt{minimizers}(X_i, l)$ is not defined **do**
        $\texttt{minimizers}(X_i, l) \leftarrow m_j$
    else
      Let l be an index such that $\texttt{minimizers}(X_i, l)$ is not defined
      Set $\texttt{ranks}(m_j, l)$ to the next unused integer in $\{1, \dots, |\mathcal{S}|\}$ in column l
      for all x' such that $m_j \in \texttt{cand}(x')$ **do**
        if $\texttt{minimizers}(x', l)$ is not defined **then**
          $\texttt{minimizers}(x', l) \leftarrow m_j$
    Remove m_j from $\texttt{cand}(X_i)$
  // Remaining ranks do not matter; complete them arbitrarily.
  for $1 \leq l \leq N$ **do**
    Assign remaining empty values in $\texttt{ranks}(\cdot, l)$ arbitrarily

For the algorithm to be correct, we must never encounter the case where there exists X_i and $m_j \in \texttt{cand}(X_i)$ such that $\texttt{minimizers}(X_i, l)$ is already defined for all $1 \leq l \leq N$. In such a case, m_j could not be a minimizer for X_i. Suppose we are in such a case. Let $\texttt{cand}_I(X_i)$ be the set of candidates of X_i at the initialization of the algorithm. Since $m_j \in \texttt{cand}_I(X_i)$ and since $\texttt{minimizers}(X_i, l)$ is defined for all l, then surely there exists $m_{j'} \in \texttt{cand}_I(X_i)$, $j' \neq j$, that appears at least twice in the list of minimizers of X_i, by pigeonhole principle ($|\texttt{cand}_I(X_i) \setminus \{m_j\}| < N$). When a minimizer is affected as a consequence of Lines 11–15, it is removed from the list of candidates (and therefore cannot be added again as a minimizer for another order), so the only way $m_{j'}$ could have been affected to two orders is as a consequence of Lines 7–9 — but those are only triggered if $m_{j'}$ is the only remaining element of $\texttt{cand}(X_i)$, thus a contradiction since it still contains at least m_j. □

D.6.3. Reduction from VertexCover

This alternative reduction was proposed by an anonymous reviewer. Let $G = (V, E)$ be a graph. Using the same formalism as above, associate to each vertex i a distinct m-mer $m_i = \texttt{binary}(i)$ encoded with $m = \lceil |V| \rceil$ bits. As above, we use the alphabet $\Sigma = \{0, 1, a\}$.

For each edge $(i, j) \in E$, assume w.l.o.g. $m_i < m_j$, we construct the k-mer $X_{(i,j)} = m_i a m_j$. We define two orders $\mathcal{O}^1$ and $\mathcal{O}^2$ as $(m_i, m_j) \in \mathcal{O}^1 \iff m_i < m_j$ and $(m_i, m_j) \in \mathcal{O}^2 \iff m_i > m_j$. Finally, we complete the orders by saying that $x < y$ if `a` appears in y and not in x, for both orders $\mathcal{O}^1$ and $\mathcal{O}^2$. If `a` belongs to both x and y, any arbitrary order can do. In the end, for all k-mers $X_{(i,j)}$, $\min_1(X_{(i,j)}) = m_i$ and $\min_2(X_{(i,j)}) = m_j$. Let $\mathcal{X} = \{X_{(i,j)} : \{i, j\} \in E, m_i < m_j\}$.

Lemma D.4. VertexCover admits a yes-instance (V, E, k) iff the MultiMinCover instance $(\mathcal{X}, \mathcal{O}^1, \mathcal{O}^2, k)$ is a yes-instance.

Proof. ($\Rightarrow$) If $\mathcal{C}$ is a vertex cover of size k, then $\mathcal{Y} = \{m_i : i \in \mathcal{C}\}$ is a multiminimizer cover: for any k-mer $X_{(i,j)}$, either $i \in \mathcal{C}$ and $\min_1(X_{(i,j)}) = m_i \in \mathcal{Y}$, or $j \in \mathcal{C}$ and $\min_2(X_{(i,j)}) = m_j \in \mathcal{Y}$.

($\Leftarrow$) Let (V, E, k) be a no-instance. Assume that our constructed instance $(\mathcal{X}, \mathcal{O}^1, \mathcal{O}^2, k)$ is a yes-instance with multiminimizer cover $\mathcal{Y}$. Then every $X_{(i,j)} \in \mathcal{X}$ has a minimizer $\min_1(X_{(i,j)}) = m_i$ or $\min_2(X_{(i,j)}) = m_j$ per definition of $\mathcal{O}^1$ and $\mathcal{O}^2$; and the set $\mathcal{C} = \{i : m_i \in \mathcal{Y}\}$ is a vertex cover of size k for $G = (V, E)$ with $E = \{(i, j) : X_{(i,j)} \in \mathcal{X}\}$. □

D.7. Density plots for multiple *w* values

In this section, we ran the same evaluation as in Section 16.4.1, but applied our multiminimizer iterator to multiple w and more m values to check that our results are robust with regard to w and m. Figure D.3 and Figure D.4 show that our results are indeed robust with regard to w: for all w values, our iterator can yield densities lower than the lower bound.

D.8. Conservation with respect to error rate

In this section, we study the evolution of the conservation of the sampled multiminimizers with respect to the error rate and the number of hash functions. Given a sequence S and a mutated version of the sequence S' for a given error rate, we measure the conservation using the Jaccard similarity of $A = \text{MultiMinimizers}(S)$ and $B = \text{MultiMinimizers}(S')$, i.e. $\frac{|A \cap B|}{|A \cup B|}$. The results are shown in Figure D.5. In particular, we can observe that the conservation decreases slightly faster with respect to the error rate as the number of hash functions increases.

D.9. Linearity of indexation time

In this section, we show that the iteration over multiminimizers is linear with regard to the number of hash functions. The results are shown in Figure D.6.

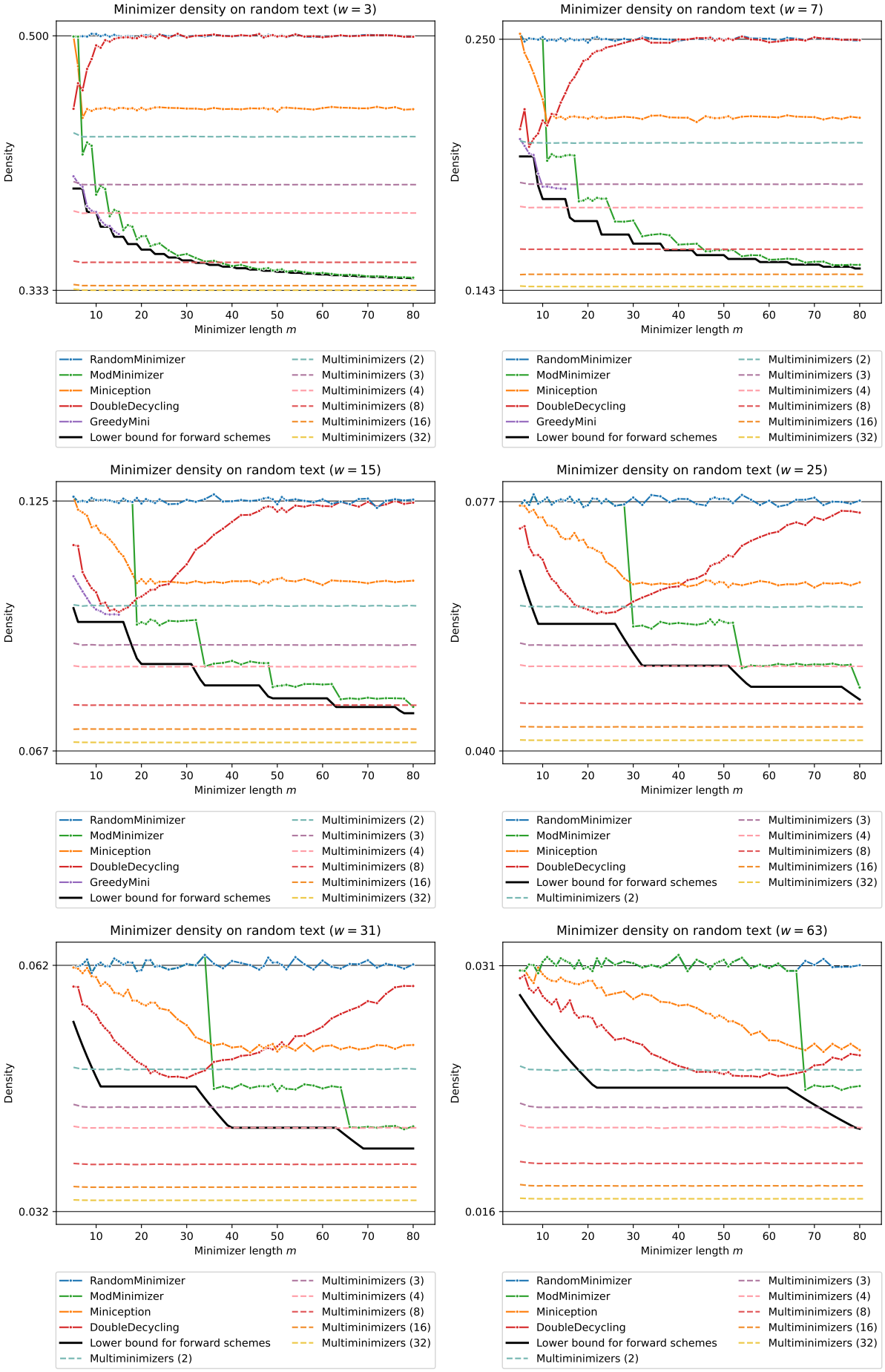


Figure D.3.: Density of multiple schemes, including multiminimizers with up to 32 hash functions, for $w \in \{3, 7, 15, 31, 63\}$.

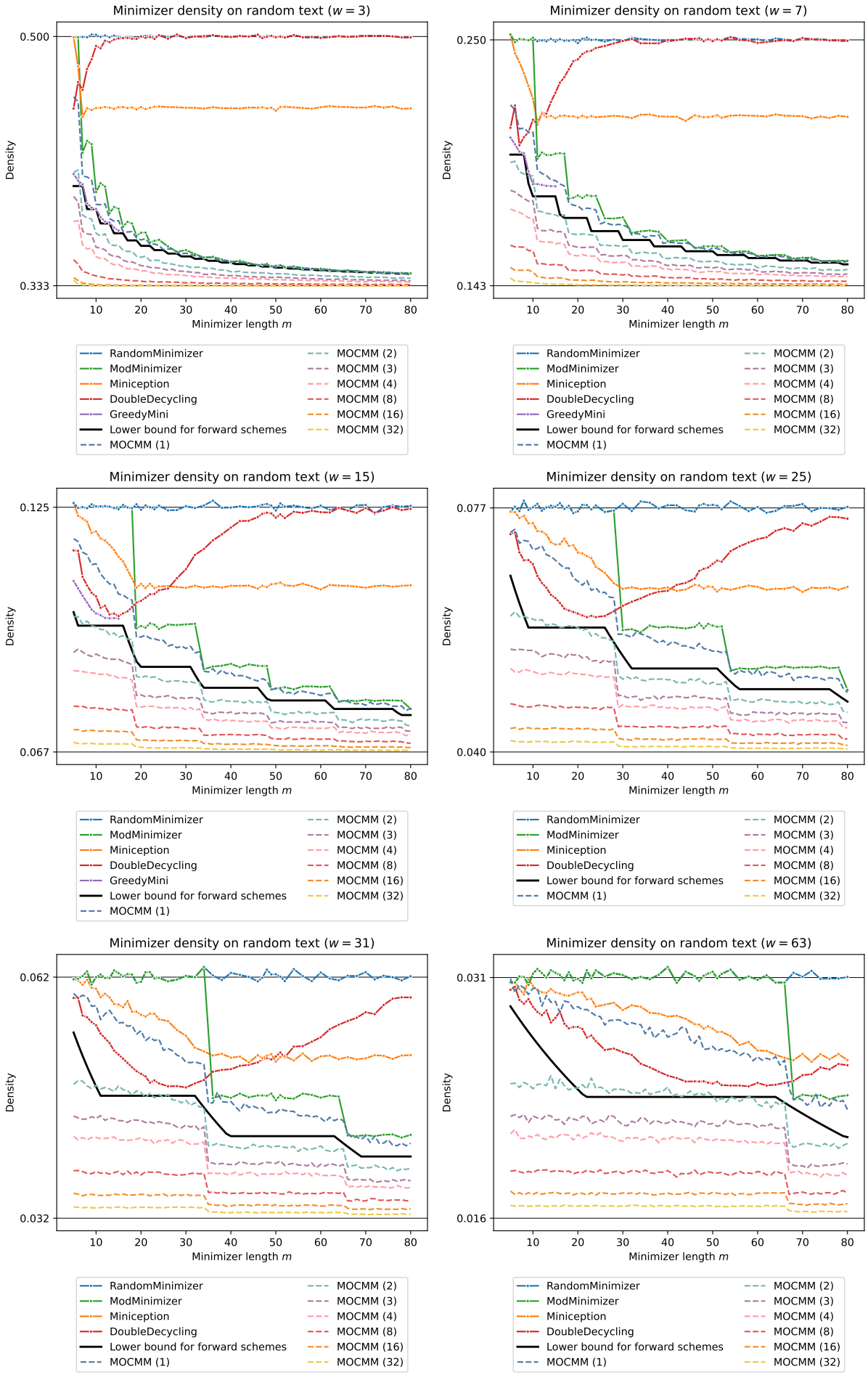


Figure D.4.: Density of multiple schemes, including multiminimizers based on open-closed mod-minimizers [GLP25] with up to 32 hash functions, for $w \in \{3, 7, 15, 31, 63\}$.

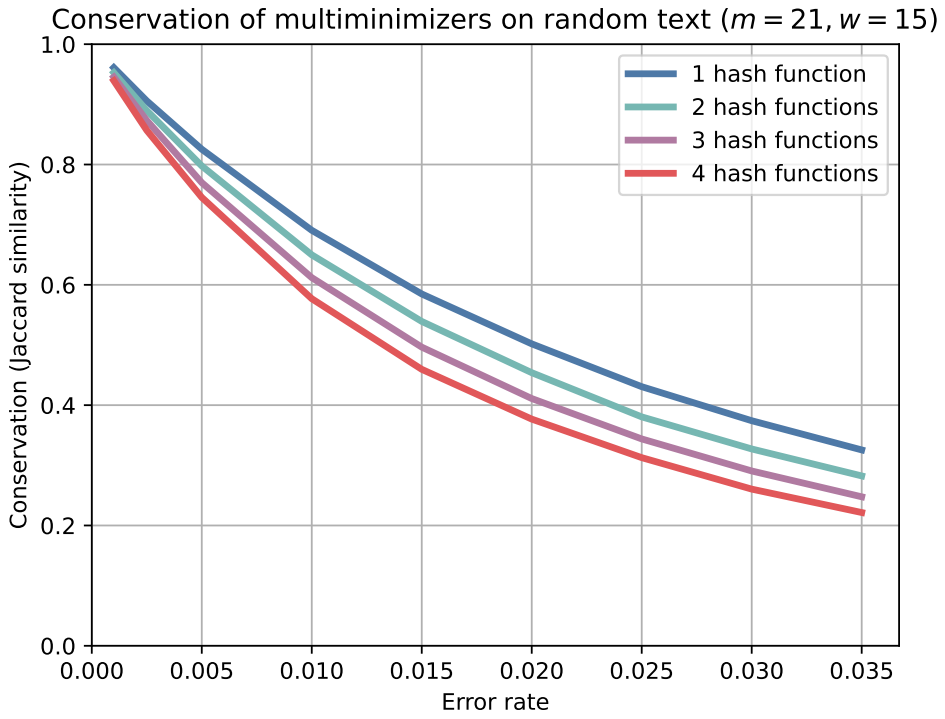


Figure D.5.: Conservation of sampled multiminimizers, measured by their Jaccard similarity, with $1 \leq N \leq 4$ for a random string of size 10^5 compared to a mutated version of the string with up to 3.5% of errors.

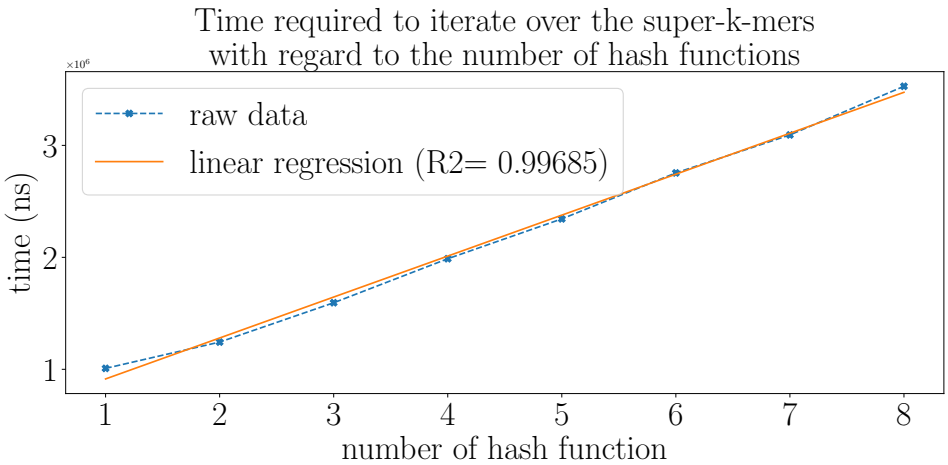


Figure D.6.: Time taken to index a random string of size 10^5. The linear regression shows that the time taken to iterate over the multiminimizers is approximately the time needed to iterate over one random minimizer scheme times the number of hash functions.

E. Proofs and experiments on fractional hitting sets

E.1. Useful lemmas

Lemma E.1. Assuming p is non-increasing with respect to w, $1-(1-p)^{w+1} \underset{w\to\infty}{=} f + o(1)$

Proof. $(1-p)^{w+1} = (1-p)^w - p(1-p)^w$

- if $p \underset{w\to\infty}{\longrightarrow} 0$, $p(1-p)^w \underset{w\to\infty}{\longrightarrow} 0$
- otherwise $p \geq c$ for some $c > 0$ since it is non-increasing, so $p(1-p)^w \leq p(1-c)^w \underset{w\to\infty}{\longrightarrow} 0$

Therefore, $(1-p)^{w+1} = (1-p)^w + o(1)$ □

We use $\widehat{\mathcal{S}}$ to denote the set of k-mers containing a small m-mer.

Lemma E.2. Given two consecutive k-mers W_0 and W_1, $\Pr(W_0, W_1 \in \widehat{\mathcal{S}}) \underset{w\to\infty}{=} f + o(1)$

Proof.

$$\begin{aligned}
\Pr(W_0, W_1 \in \widehat{\mathcal{S}}) &= 1 - \Pr(W_0 \notin \widehat{\mathcal{S}} \vee W_1 \notin \widehat{\mathcal{S}}) \\
&= 1 - \left[\Pr(W_0 \notin \widehat{\mathcal{S}}) + \Pr(W_1 \notin \widehat{\mathcal{S}}) - \Pr(W_0 \notin \widehat{\mathcal{S}} \wedge W_1 \notin \widehat{\mathcal{S}})\right] \\
&= 1 - 2(1-p)^w + (1-p)^{w+1} = 1 - (1-p)^w + o(1) \\
&= f + o(1)
\end{aligned}$$

□

Lemma E.3. $p \underset{w\to\infty}{=} -\frac{1}{w}\ln(1-f) + o(1/w)$

Proof. Because of Proposition 17.1, we have $p = 1-(1-f)^{1/w}$ and

$$(1-f)^{1/w} = \exp\left[\frac{1}{w}\ln(1-f)\right] = 1 + \frac{1}{w}\ln(1-f) + o(1/w)$$

□

E.2. Proof of Theorem 17.1

In order to upper bound the density, we follow the same approach as the one presented in [ZKM20] (for the proof of theorem 7). As stated in [ZKM20], the density is equivalent to the probability that a context c (that is, the string formed by two consecutive k-mers) is *charged*, i.e. the two k-mers of c have different minimizers.

$$\begin{aligned}
d &= \Pr_{c,h}(c \text{ is charged}) \\
&\leq \Pr_{c,h}(c \text{ has duplicate } m\text{-mers}) + \Pr_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers})
\end{aligned}$$

Lemma E.4 (lemma 9 from [ZKM20])**.** Assuming $m > (3+\varepsilon)\log_\sigma w$,

$$\Pr_{c,h}(c \text{ has duplicate } m\text{-mers}) = o(1/w)$$

If c has no duplicate m-mers, the small m-mers are all distinct and each of them has the same probability to be minimal since h is random. Therefore,

$$\Pr_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers}) = \mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right]$$

where $M_{boundary}$ denotes the number of boundary m-mers that are small and M_{total} denotes the total number of small m-mers in c.

Let x_0 denote the first m-mer of c and x_w denote the last one,

$$\begin{aligned}\mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right] &= \mathbb{E}_{c,h}\left[\frac{\mathbf{1}_{x_0\in\mathcal{S}} + \mathbf{1}_{x_w\in\mathcal{S}}}{M_{total}}\right] = 2\cdot\mathbb{E}_{c,h}\left[\frac{\mathbf{1}_{x_0\in\mathcal{S}}}{M_{total}}\right] \quad \text{(symmetry)}\\ &= 2\cdot\mathbb{E}_{c,h}\left[1/M_{total} \mid x_0\in\mathcal{S}\right]\cdot\Pr(x_0\in\mathcal{S})\end{aligned}$$

Assuming x_0 is small, we have $M_{total} = 1 + X$ with $X \sim B(w,p)$, since each other m-mer of c has a probability p to be small.

$$\mathbb{E}_{c,h}\left[1/M_{total} \mid x_0\in\mathcal{S}\right] = \mathbb{E}_{c,h}\left[\frac{1}{1+X}\right] = \sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i}$$

Lemma E.5. $\sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i} = \frac{1-(1-p)^{w+1}}{(w+1)p}$

Proof.

$$\begin{aligned}(w+1)p\sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i} &= \sum_{i=0}^{w}\frac{w+1}{1+i}\frac{w!}{i!(w-i)!}p^{i+1}(1-p)^{w-i}\\ &= \sum_{i=0}^{w}\frac{(w+1)!}{(i+1)!(w+1-(i+1))!}p^{i+1}(1-p)^{w+1-(i+1)} = \sum_{j=1}^{w+1}\binom{w+1}{j}p^j(1-p)^{w+1-j}\\ &= 1-(1-p)^{w+1}\end{aligned}$$

□

Finally, since $\Pr(x_0\in\mathcal{S}) = p$,

$$d \le 2\cdot\frac{1-(1-p)^{w+1}}{w+1} + o(1/w) = \frac{2f}{w+1} + o(1/w)$$

E.3. Proof of Theorem 17.2

In this section, **we assume that every *k*-mer we work with contains a small m-mer**.

Just as for the proof of Theorem 17.1, we still have

$d \le \Pr_{c,h}(c \text{ has duplicate } m\text{-mers}) + \Pr_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers})$

and

$$\begin{aligned}\Pr_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers}) &= \mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right]\\ &= 2\cdot\mathbb{E}_{c,h}\left[1/M_{total} \mid x_0\in\mathcal{S}\right]\cdot\Pr(x_0\in\mathcal{S} \mid W_1\in\widehat{\mathcal{S}})\end{aligned}$$

Lemma E.6. Assuming $m > (3+\varepsilon)\log_\sigma w$, $\Pr_{c,h}(c \text{ has duplicate } m\text{-mers}) = o(1/w)$

Proof. This proof is similar to the proof of lemma 9 from [ZKM20]. Let $i, j \in [\![0, w]\!]$ with $i < j$, $\delta = j - i$.

If $\delta < m$, $\Pr(x_i = x_j) = \frac{\sigma^\delta}{\sigma^{m+\delta}} = \frac{1}{\sigma^m} = o(1/w^3)$

If $\delta \geq m$,

$$\begin{aligned}\Pr(x_i = x_j) &= \Pr(x_i = x_j \mid x_i, x_j \in \mathcal{S}) \Pr(x_i, x_j \in \mathcal{S}) + \Pr(x_i = x_j \mid x_i, x_j \notin \mathcal{S}) \Pr(x_i, x_j \notin \mathcal{S}) \\ &= \frac{\Pr(x_i, x_j \in \mathcal{S})}{p \cdot \sigma^m} + \frac{\Pr(x_i, x_j \notin \mathcal{S})}{(1-p)\sigma^m}\end{aligned}$$

Because of Lemma E.2, $\Pr(x_i, x_j \in \mathcal{S}) = \frac{p^2}{\Pr(W_0, W_1 \in \widehat{\mathcal{S}})} = \frac{p^2}{f + o(1)} \leq p$ and

$$\Pr(x_i, x_j \notin \mathcal{S}) \leq \frac{(1-p)^2 \left[1 - (1-p)^{w-1}\right]}{\Pr(W_0, W_1 \in \widehat{\mathcal{S}})} = \frac{(1-p)^2 \left[1 - (1-p)^{w-1}\right]}{f + o(1)} \leq (1-p)^2$$

Therefore, $\Pr(x_i = x_j) \leq \frac{p}{p \cdot \sigma^m} + \frac{(1-p)^2}{(1-p)\sigma^m} \leq \frac{2}{\sigma^m} = o(1/w^3)$

Thus $\Pr_{c,h}(c \text{ has duplicate } m\text{-mers}) = \binom{w}{2} \times o(1/w^3) = o(1/w)$ □

Assuming x_0 is small, the w next m-mers of c form a k-mer, so we know that at least one of them is also small. Therefore,

$$\begin{aligned}&\mathbb{E}_{c,h}\left[1/M_{total} \mid x_0 \in \mathcal{S}\right] = \mathbb{E}_{c,h}\left[\frac{1}{1+X} \mid X \geq 1\right] = \frac{1}{\Pr(X \geq 1)} \sum_{i=1}^{w} \frac{1}{1+i} \binom{w}{i} p^i (1-p)^{w-i} \\ &= \frac{1}{f}\left[\frac{1 - (1-p)^{w+1}}{(w+1)p} - (1-p)^w\right] = \frac{1}{f}\left[\frac{f + o(1)}{(w+1)p} - (1-f)\right]\end{aligned}$$

What's more, $\Pr(x_0 \in \mathcal{S} \mid W_1 \in \widehat{\mathcal{S}}) = \frac{\Pr(x_0 \in \mathcal{S})}{\Pr(W_1 \in \widehat{\mathcal{S}})} = \frac{p}{f}$.

Hence,

$$\begin{aligned}d &\leq \frac{2p}{f^2}\left[\frac{f + o(1)}{(w+1)p} - (1-f)\right] + o(1/w) = \frac{2}{f(w+1)} - \frac{2(1-f)p}{f^2} + o(1/w) \\ &= \frac{2}{f(w+1)} + \frac{2(1-f)\ln(1-f)}{f^2 w} + o(1/w) \\ &= 2 \cdot \frac{f + (1-f)\ln(1-f)}{f^2(w+1)} + o(1/w)\end{aligned}$$

E.4. Proof of Theorem 17.3

In order to compute the proportion of maximal super-k-mers, we adapt the proof of theorem 4 from [PSL23].

First, we introduce a similar Markov chain representing the position X of the small minimizer in the k-mer, with an extra state $\emptyset$ when there is no small minimizer.

We reuse the following notations introduced in [PSL23]:

- P_{lr} is the proportion of left-right-max (i.e. maximal) super-k-mers
- P_l is the proportion of left-max super-k-mers
- P_r is the proportion of right-max super-k-mers
- P_n is the proportion of non-max super-k-mers

$$\forall i \in [\![1, w-1]\!], \Pr(\text{first } X = i) = \Pr(X = 1) \cdot \frac{f}{w}$$

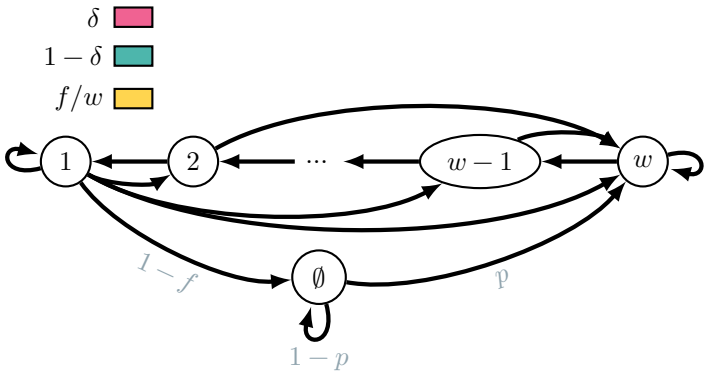


Figure E.1.: The chain is in state $i \in [\![1, w]\!]$ if the small minimizer starts at position i in the k-mer, and $\emptyset$ if there is no small minimizer. Different edge colors represent different probabilities.

$$P_{lr}+P_r = \Pr(X=w) = \Pr(\text{first } X=w) = 1-\sum_{i=1}^{w-1}\Pr(\text{first } X=i) = 1-\Pr(X=1)\cdot f\cdot(1-1/w)$$

$$P_{lr}+P_l = \Pr(\text{last } X=1) = \Pr(X=1)+\Pr(\text{first } X=1) = \Pr(X=1)\cdot(1+f/w)$$

By symmetry, $P_l = P_r$, so $1-\Pr(X=1)\cdot f\cdot(1-1/w) = \Pr(X=1)\cdot(1+f/w)$

Therefore, $\Pr(X=1) = \frac{1}{1+f}$ and $\Pr(X=w) = 1-\frac{f}{1+f}\left(1-\frac{1}{w}\right) = \frac{1+f/w}{1+f}$

What's more, $1+P_{lr} = P_{lr}+P_l+P_{lr}+P_r+P_n = 2\cdot\Pr(X=w)+P_n$

so

$$P_{lr} = P_n+2\cdot\Pr(X=w)-1 = P_n+\frac{1-f(1-2/w)}{1+f} \text{ and } P_l = P_r = \Pr(X=w)-P_{lr} = \frac{f(1-1/w)}{1+f}-P_n$$

$$P_n = \Pr(\text{first } X\neq w)\cdot\Pr(\text{last } X\neq 1) = \Pr(X=1)\cdot f\cdot(1-1/w)\cdot[1-\Pr(X=1)\cdot(1+f/w)]$$
$$= \left(1-\frac{1}{w}\right)\cdot\frac{f}{1+f}\cdot\left[1-\frac{1+f/w}{1+f}\right] = \left(1-\frac{1}{w}\right)\cdot\frac{f}{1+f}\cdot\frac{f(1-1/w)}{1+f} = \left[\left(1-\frac{1}{w}\right)\frac{f}{1+f}\right]^2$$

Thus $P_l = P_r = \left[\left(1-\frac{1}{w}\right)\frac{f}{1+f}\right]\left[1-\left(1-\frac{1}{w}\right)\frac{f}{1+f}\right]$

and

$$P_{lr} = \left[\left(1-\frac{1}{w}\right)\frac{f}{1+f}\right]^2+\frac{1-f(1-2/w)}{1+f}$$

E.5. Proof of Theorem 17.4

This proof generalizes the proof of Theorem 17.1 when the minimizers are selected from a UHS $\mathcal{U}$ with density $d_{\mathcal{U}}$.

First, because of independence, we have $\Pr(x_0 \in \mathcal{S}\cap\mathcal{U}) = \Pr(x_0\in\mathcal{S})\cdot\Pr(x_0\in\mathcal{U})$ and $\Pr(|\mathcal{S}\cap\mathcal{U}\cap c| = i) = \sum_{n\geq i}\Pr(|\mathcal{U}\cap c| = n)\Pr(|\mathcal{S}\cap\mathcal{U}\cap c| = i \mid |\mathcal{U}\cap c| = n)$

The main change of the proof lies in the bound on the expectation:

$$\begin{aligned}
&\mathbb{E}\left[\frac{1}{|\mathcal{S}\cap\mathcal{U}\cap c|} \mid x_0\in\mathcal{S}\cap\mathcal{U}\right] = \sum_{i=0}^{w}\frac{1}{i+1}\Pr\left(|\mathcal{S}\cap\mathcal{U}\cap c|=i+1 \mid x_0\in\mathcal{S}\cap\mathcal{U}\right)\\
&=\sum_{i=0}^{w}\frac{1}{i+1}\sum_{n=i}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\Pr(|\mathcal{S}\cap\mathcal{U}\cap c|=i+1\mid|\mathcal{U}\cap c|=n+1,x_0\in\mathcal{S}\cap\mathcal{U})\\
&=\sum_{n=0}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\sum_{i=0}^{n}\frac{1}{i+1}\Pr(|\mathcal{S}\cap\mathcal{U}\cap c|=i+1\mid|\mathcal{U}\cap c|=n+1,x_0\in\mathcal{S}\cap\mathcal{U})\\
&=\sum_{n=0}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\sum_{i=0}^{n}\frac{1}{i+1}\Pr(|\mathcal{S}|=i+1\mid x_0\in\mathcal{S},|W|=n)\\
&=\sum_{n=0}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\cdot\frac{1-(1-p)^{n+1}}{(n+1)p}\\
&\leq\sum_{n=0}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\cdot\frac{f+o(1)}{(n+1)p}
\end{aligned}$$

Therefore, using the same arguments as in the proof of Theorem 17.1, we obtain

$$\begin{aligned}
d_{\mathcal{S}\cap\mathcal{U}} &\leq 2\cdot\Pr(x_0\in\mathcal{S}\cap\mathcal{U})\cdot\mathbb{E}\left[\frac{1}{|\mathcal{S}\cap\mathcal{U}\cap c|}\mid x_0\in\mathcal{S}\cap\mathcal{U}\right]+o(1/w)\\
&\leq 2\cdot\Pr(x_0\in\mathcal{U})\cdot\Pr(x_0\in\mathcal{S})\sum_{n=0}^{w}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})\cdot\frac{f}{(n+1)p}+o(1/w)\\
&= f\cdot 2\cdot\Pr(x_0\in\mathcal{U})\sum_{n=0}^{w}\frac{1}{n+1}\Pr(|\mathcal{U}\cap c|=n+1\mid x_0\in\mathcal{U})+o(1/w)\\
&= f\cdot d_{\mathcal{U}}+o(1/w)
\end{aligned}$$

E.6. Additional figures

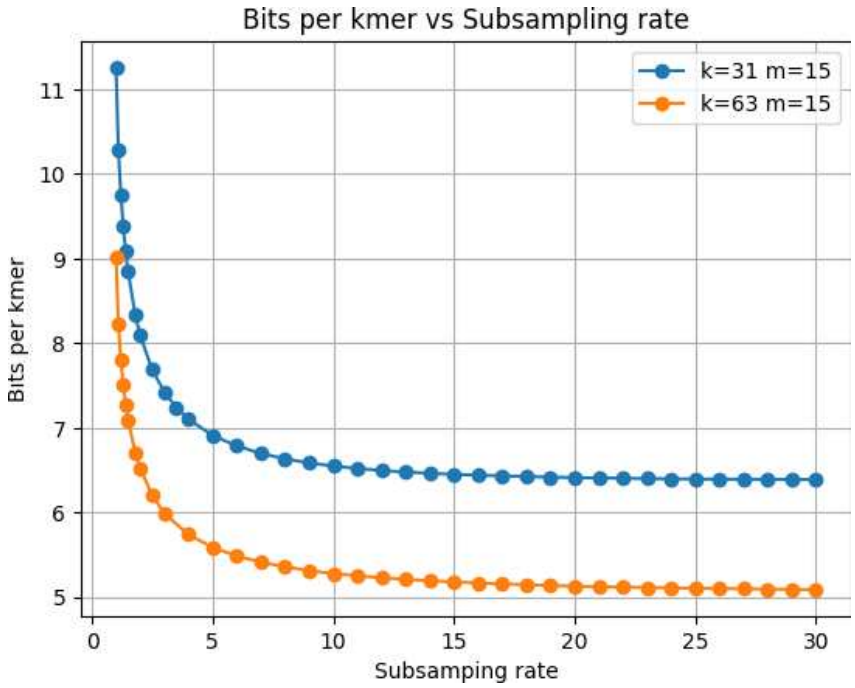


Figure E.2.: Space cost in bits per k-mer according to the subsampling rate.

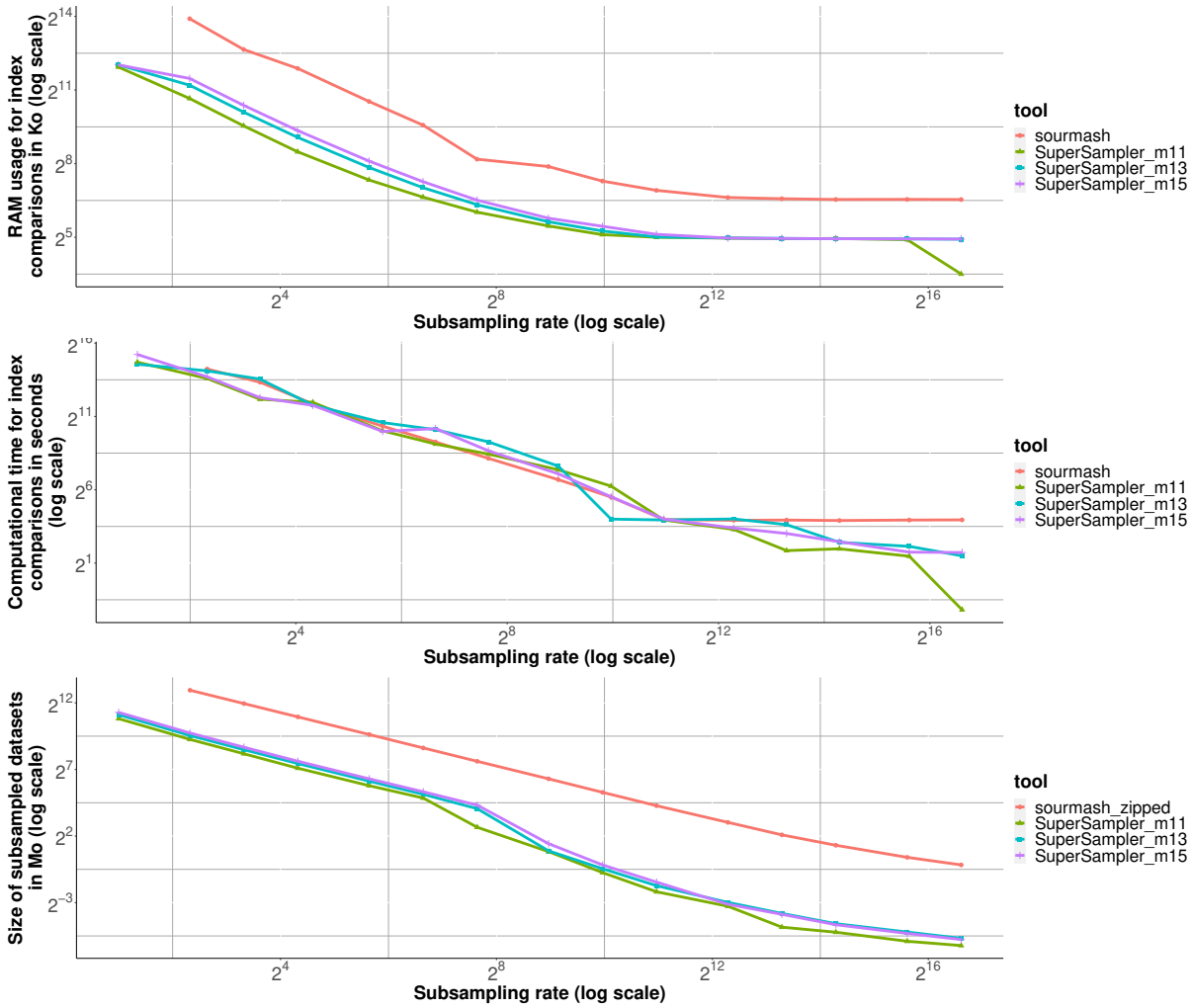


Figure E.3.: Salmonellas 1K k=31

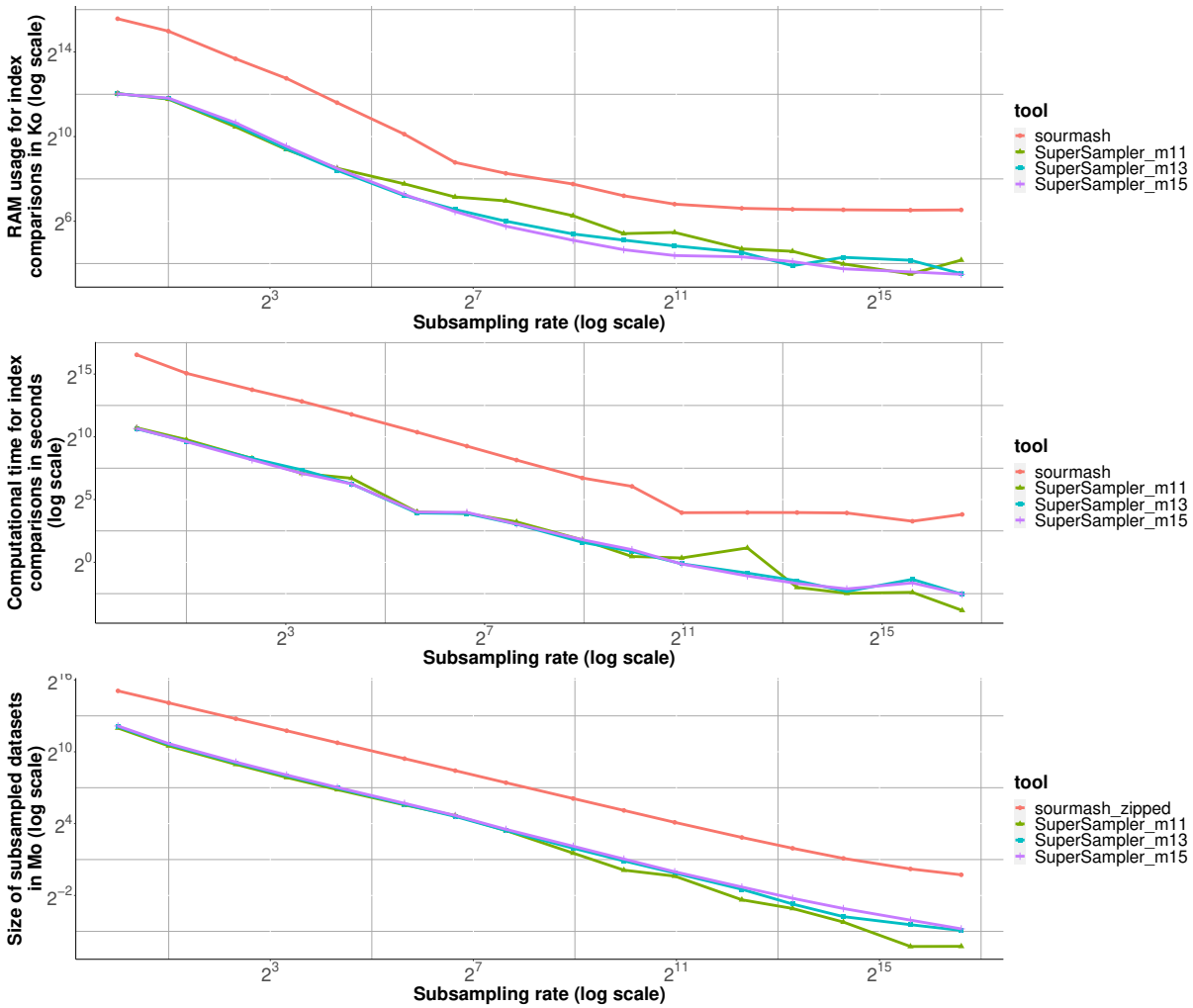


Figure E.4.: Refseq 1K k=63

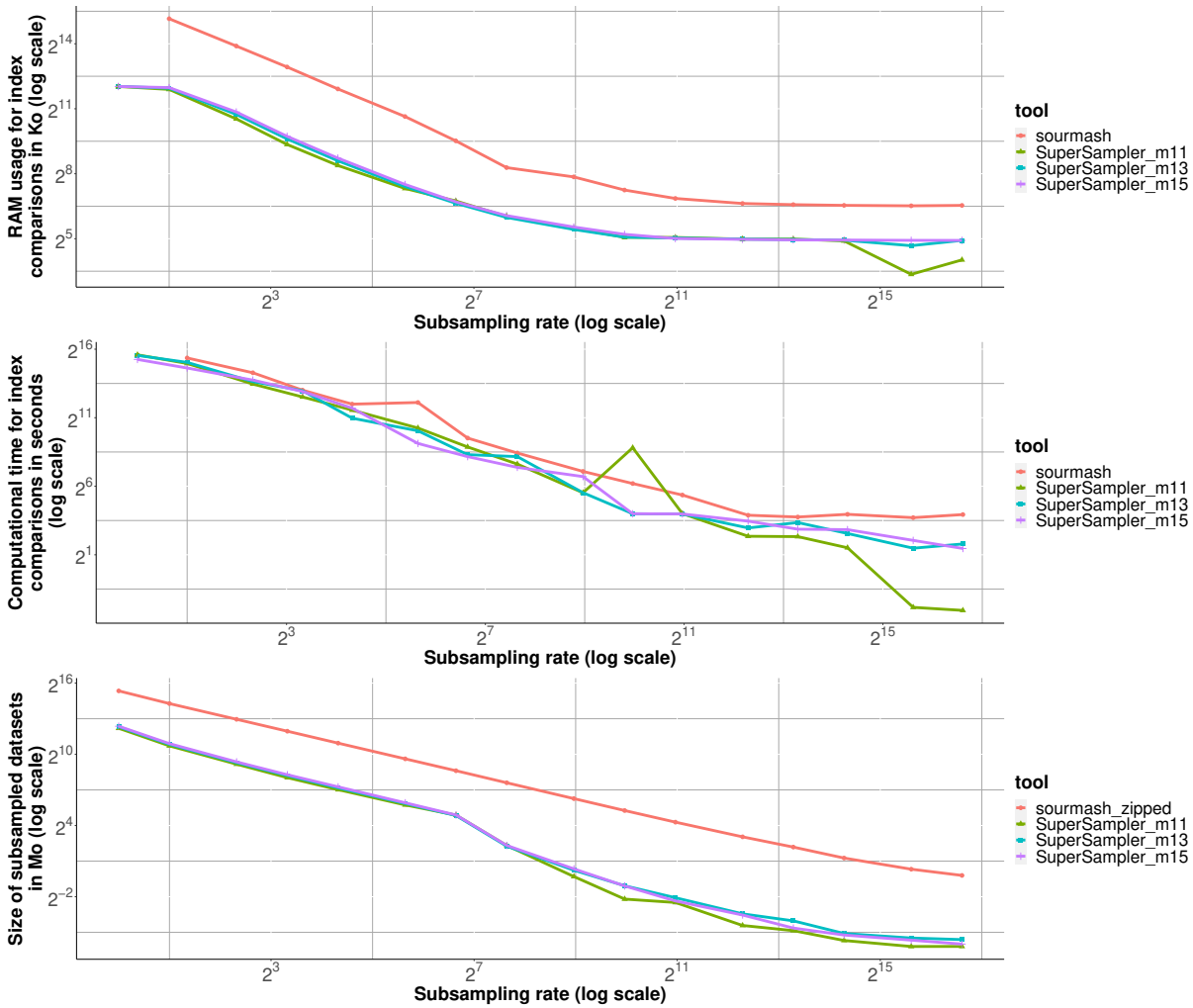


Figure E.5.: Salmonellas 1K k=63

F. Useful tools

I wanted to take this thesis as an opportunity to share some tools that I found particularly useful during my PhD.

F.1. Typesetting this thesis

I decided early on to use Quarto to typeset this thesis. Looking back, I think this was a great choice for multiple reasons:

- I find its (enriched) Markdown syntax to be much nicer to use than LaTeX, and it helped me focus on the content itself.
- Having a live web version that gets instantly updated as you type is incredibly convenient compared to the long compilation time of LaTeX.
- It still provides fine-grained control of figures and references, and supports many themes and extensions.
- Because it is built on top of Pandoc, it can export a wide variety of formats, which is what let me have a web version and a PDF produced with the same sources.

You can find all the sources of this thesis alongside the extensions I tweaked at https://github.com/imartayan/phd.

F.2. Bibliography management and formatting

The main solution I use to manage my collection of articles is Zotero. I find it easy to use, and its built-in PDF annotation tool is very convenient. If you feel limited by its storage sync limit of 300 MB, note that you can actually use any WebDAV server to sync your files instead. For instance, if your institution has a Nextcloud server, you can use it to sync larger collections. They also recently made an iOS/Android app to access your collection, which can be useful if you prefer using a tablet.

While writing this thesis, I very often needed to add BibTeX entries for new articles. At first I was using Google Scholar to do that, but I quickly noticed that its entries were often incomplete and that it systematically stripped DOIs, which is the most important information in my opinion. Thus I decided to write my own tool called bibelot (*bib a lot*) to do that. It's a CLI tool that takes DOIs or preprint links as input, fetches up-to-date metadata, and copies the resulting BibTeX entries to a file or to your clipboard.

F.3. CLI tools I use everyday

I spend most of my time working from a terminal, and there are a few tools that I couldn't live without:

- First for the terminal itself, Ghostty is a very nice piece of software, fast and very polished.
- When working on a remote server, bottom is a great replacement for `top` to monitor processes and resource usage.
- If you need to monitor disk usage, dust is a great replacement for `du`.

- When searching for a pattern inside text files, ripgrep is usually much faster than `grep`.
- Ripgrep can actually be extended with ripgrep-all to search many other formats such as PDF. This allows me to search for a keyword in my entire collection of articles in a couple of seconds.

F.4. Rust toolchain

These days, I write most of my code in Rust and in addition to the built-in toolchain, there are a few more tools that I find very convenient:

- While the warning messages of the compiler are already very useful, clippy provides even more suggestions and can even automatically apply fixes. You can also set up your favorite editor to integrate clippy warnings to your code.
- I don't always notice when my dependencies are outdated, and cargo-outdated can check for that automatically.
- Similarly, cargo-machete lets you remove unused dependencies.
- Flame graphs are a great way to visualize the trace of your program to identify which functions dominate the running time, you can easily generate them with cargo-flamegraph.
- Sometimes flame graphs are not enough and you want to look at the assembly generated for a specific function, this can be done with cargo-show-asm. This is especially useful to understand what the compiler is doing behind your back.
- Finally, if you end up publishing crates, cargo-release provides many sanity checks before actually doing the release.

You can install all these tools using the following commands:

```
rustup update
rustup component add clippy
cargo install cargo-outdated cargo-machete flamegraph cargo-show-asm cargo-release
```