Finimap: Fast and Accurate Single-Species Bacterial Pseudoalignment with Finimizers

Jarno N. Alanko
Department of Computer Science, University of Helsinki, Finland

Elena Biagi
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Abstract

In recent years, pseudoalignment as a means for mapping reads to databases of reference genomes has become a widely-used method in studies of bacterial pathogenesis. A popular pseudoalignment criterion is thresholded union, in which a read is said to pseudoalign to a reference if the reference contains more than a percentage t of the read's k-mers. Several pseudoalignment indexing tools that implement this and other pseudoalignment criteria are now available, including Bifrost, Themisto, and Fulgor. In this paper, we describe a scheme for single-species bacterial pseudoalignment that, instead of k-mers, uses shortest unique finimizers (Alanko et al., IEEE/ACM TCBB, 2025) as features for determining pseudoalignment. We show that this scheme, which we call Finimap, leads to a significantly lower false-positive rate than other recent "approximate pseudoalignment" methods Kaminari and Raptor, and is also faster.

2012 ACM Subject Classification Applied computing → Bioinformatics; Theory of computation → Data structures design and analysis; Theory of computation → Pattern matching; Theory of computation → Algorithm design techniques; Applied computing → Computational genomics

Keywords and phrases Pseudoalignment, sequence alignment, approximate string matching, string processing, k-mer, data structures, data compression

Digital Object Identifier 10.4230/LIPIcs.WABI.2026.27

Supplementary Material *Software (Source Code)*: `https://github.com/ElenaBiagi/Finimap` archived at `swh:1:dir:b8b539b0f246d45853d327fd276ad4cd9efe7f7e`

Funding This work was supported by the Research Council of Finland via grant 339070.
Elena Biagi: The Finnish Cultural Foundation (grant 00250241) and the Finnish Accademy of Science and Letters (Vilho, Yrjö and Kalle Väisälä Fund).

Acknowledgements The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Pseudoalignment is an approximate form of sequence alignment in which a match between two sequences is quantified as a function of their shared k-mer content. Typically, pseudoalignment is performed between a sequence read and an indexed collection of reference genomes. Threshold union is a popular form of pseudoalignment in which all k-mers of the query read are searched in the index, and a match is reported if at least a fraction τ of the matched k-mers occurs in a reference genome. Unlike with traditional sequence alignment, the precise genomic coordinates of the match are not reported.

Computationally, pseudoalignment is much cheaper than regular alignment (as computed by tools such as, e.g., minimap [22]) and retains accuracy at a level that is sufficient for many downstream applications. Pseudoalignment originated in RNA-seq quantification [8] as a way to remove a computational bottleneck in aligning reads. It has since found wide

use in the study of bacterial genomics [5, 20, 24, 25] and the problem of metagenomic read assignment [35, 32]. To highlight just two examples: Khawaja et al. [20] use pseudoalignment (in particular the Themisto tool [5]) in the study of diversity and antibiotic resistance in local populations of *E. coli*; and Mäklin et al. [24] based their mGEMS pipeline for genomic epidemiology – a system for tracing transmission of target pathogens – on pseudoalignment. *The single-species bacterial pangenome use case is our focus throughout this paper.*

There now exist several efficient k-mer-based pseudoalignment indexing tools, including Themisto [5], Fulgor [9, 10, 12, 30], Metagraph [19], and Bifrost [15]. Recently, there has been a move toward pseudoalignment methods that are not based on k-mers which further loosen match criteria with the aim of further reducing query times. These tools can be thought of as approximating the results of k-mer based pseudoalignment, even if there are no formal approximation guarantees. Raptor [37] and Kaminari [21], are two such methods that make use of *minimizers* [34, 33, 36] instead of k-mers as features. This possibly makes their matches less precise than the k-mer based pseudoalignment tools, but their operations faster.

Contribution. In this paper, we introduce a new (approximate) method for pseudoalignment based on shortest unique *finimizers* [2], variable length substrings that are unique in the de Bruijn graph of the indexed data set (we give a formal definition below). Our tool, Finimap, indexes and queries only finimizers instead of full k-mers, with color sets of k-mers sharing the same finimizer merged during indexing. In particular, we present the following results:

1. We demonstrate that Finimap is fast compared to existing approximate methods Raptor and Kaminari, while producing drastically fewer false positive matches relative to established k-mer based pseudoalignment tools. For example, when pseudoaligning 50,000 long reads to a set of 3682 *E. coli* genomes, Finimap had a false positive rate of 2% and ran in 30 seconds, compared to 15% by Raptor, which had a similar run time. Kaminari had a false positive rate of 13% and was 2× slower at its fastest setting.
2. In order to explain the superior fidelity of Finimap compared to the minimizer-based Raptor and Kaminari schemes, we elucidate an interesting property of finimizers, namely, that they partition the de Bruijn graph into connected components in a way that minimizers do not. This property may be of independent interest.
3. Since finimizers have variable length, they pose distinct challenges compared to k-mers and minimizers (used by Raptor and Kaminari), particularly during the lookup phase. To address this, we propose a novel indexing scheme based on Elias–Fano encoding, that is fast and requires a modest amount of space relative to the total index size.

Roadmap. In the next section we set notation and define basic concepts used throughout. Then, in Section 3, we review prior work on indexing for pseudoalignment. Section 4 introduces Finimap, our finimizer-based pseudoalignment index and Section 5 presents experiments measuring its performance. In Section 6 we delve more deeply into properties of finimizers, searching for an explanation for Finimap's superior accuracy compared to minimizer-based pseudoalignment. Conclusions and avenues for future work are then offered.

2 Preliminaries

Throughout the paper, a *string* $X[1..n]$ defines a sequence of $|X|$ symbols. Since we deal with bioformatics applications, our alphabet is the DNA alphabet, which consists of four symbols $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. The empty string is denoted ϵ and $|\epsilon| = 0$. The *substring* of X denoted $X[i..j]$ starts at symbol i and ends at symbol j. We also use the half-open interval notation $X[i..j) = X[i..j-1]$ and $X(i..j] = X[i+1..j]$. A *prefix* is a substring starting at position 1 and a *suffix* is a substring ending at position n. A *k-mer* is a (sub)string of length k.

▶ **Definition 1** (k-Spectrum). *The k-spectrum $S_k(X)$ of a string X is the set of distinct k-mers $\{X[i..i+k-1] \mid i = 1, \ldots, |X| - k + 1\}$. The k-spectrum $S_k(X_1, \ldots, X_m)$ of a set of strings $X_1, \ldots, X_m$ is the union $\bigcup_{i=1}^{m} S_k(X_i)$*

▶ **Definition 2** (Padded k-Spectrum). *Let $R = S_k(X_1, \ldots, X_m)$ be a k-spectrum with alphabet Σ, and let $R' \subseteq R$ be the set of k-mers Y such that $Y[1..k-1]$ is not present as a suffix of any k-mer in R. The padded k-spectrum is the set $S_k^+(X_1, \ldots, X_m) = S_k \cup \{\$^k\} \cup \bigcup_{Y \in R'} \{\$^{k-i} Y[1..i] \mid i = 1, \ldots, k-1\}$, with special character $\$ \notin \Sigma$ and $\$ < c$ for all $c \in \Sigma$.*

In a *padded k-spectrum* a minimal set of \$-padded *dummy* k-mers are added to ensure that in the de Bruijn graph of $S_k(X_1, \ldots, X_m)$ and the dummy k-mers, every non-dummy k-mer has an incoming path of length at least k. Here, we use a padded node-centric definition of a de Bruijn Graph:

▶ **Definition 3** (de Bruijn Graph). *The de Bruijn graph of a set of strings $X_1, \ldots, X_m$ is a directed graph $G = (V, E)$, where each vertex represesents a k-mer: $V = S_k^+(X_1, \ldots, X_m)$. And there exists a directed edge from vertex i to vertex j, representing k-mers u and v, repsectuvely, iff $u(1..k] = v[1..k)$.*

The label of an edge $e = (i, j)$, denoted with $\ell(e)$ or $\ell(i, j)$, is the character $v[k]$. The label of a path of edges is the concatenation of the edge labels in the order of the path.

We now define a central notion of this paper: *finimizers*, introduced by Alanko et al. [2].

▶ **Definition 4** (Shortest t-Finimizers). *Let G be the de Bruijn graph of a set of k-mers R, X be a k-mer, and Y be a substring of X, and $t \geq 1$ be an integer. We call Y the* shortest t-finimizer *of X with respect to the k-mer set R if Y occurs at most t times in G – t nodes in G are reached by a path spelling Y – and there is no shorter substring of X with at most t occurrence in G.*

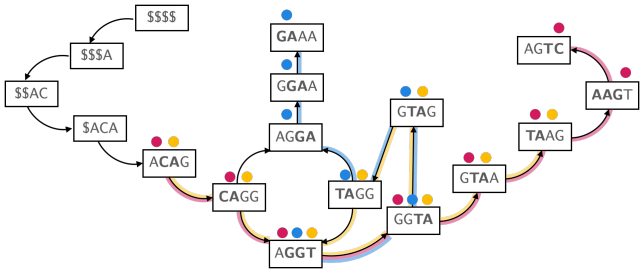
If multiple shortest unique finimizer candidates exist, a consistent tiebreaker such as lexicographic comparison can be used to always pick the same finimizer.

When $t = 1$, the finimizers are called *Shortest Unique Finimizers (SUFs)*. With a careful implementation, a finimizer function for shortest unique finimizers can be constructed in time linear in the total length of the input k-mer set [2], and evaluated for every k-mer of a query sequence Q in time $O(|Q|)$. Note that a *SUF* exists for every k-mer in the index as every k-mer appears only once in the dDG. From now on, with a little abuse of notation, when mentioning finimizers we will always refer to *SUF*s. In Figure 1b, finimizers are in bold: the finimizer of AGGT is GGT and not GT as there are two paths on the graph labeled with GT, namely the paths ACAG-AGGT and GTAA-AAGT, but only one path labeled GGT: \$ACA-AGGT.

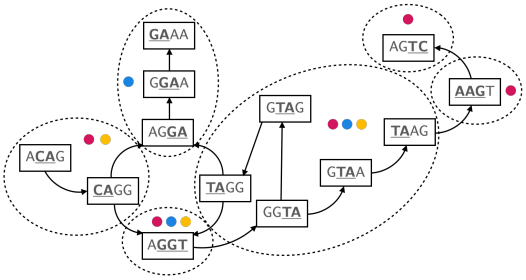
When indexing a set of genomes, we deal with *colored de Bruijn graphs*, where each node represents a k-mer from a reference collection, and is labeled with a color indicating the set of references in which the k-mer appears (see Figure 1a).

As mentioned in the introduction, there exist several previously published tools [21, 37, 38] that solve pseudoalignment approximately use the concept of minimizers [34, 33, 36]. For completeness we define minimizers now. A minimizer is the smallest m-mer of k-mer:

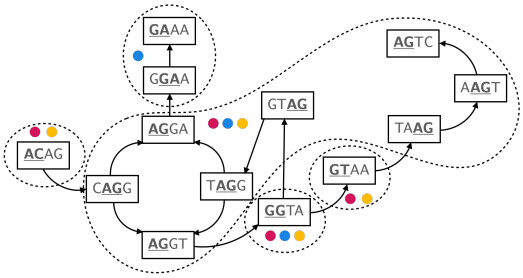
[1] For the sake of this example AA has not been considered a valid lexicographic minimizer as it is usually the case with stretches of As.



(a) Colored de Bruijn graph of $\mathcal{T}$.



(b) Colored dBG of $\mathcal{T}$ with highlighted finimizer partitions.



(c) Colored dBG of $\mathcal{T}$ with highlighted minimizer partitions.

Figure 1 (a) Colored de Bruijn graph, with $k = 4$, of the set of genomes $\mathcal{T} = \{$ACAGGTAAGTC, ACAGGTAGGTAAG, AGGTAGGAAA$\}$. Nodes are colored based on the genomes in which they appear, their colorset. (b) Color sets for 4-mers sharing the same finimizer ($t = 1$), in bold un underlined, are merged. (c) Color sets for 4-mers sharing the same lexicographic minimizer[1] ($m = 2$), shown in bold and underlined, are merged.

▶ **Definition 5** (Minimizer). *Let X be a k-mer, and m be an integer such that $m \leq k$. The minimizer of X is the smallest m-mer of X according to a given order relation of the minimizers.*

The chosen order relation could be the simple lexicographic order or, more commonly, something more complex, e.g., based on hashing.

3 Pseudoalignment Indexes

Conceptually, indexes for k-mer based pseudoalignment associate a color set with each indexed k-mer. The color set associated with a given k-mer is a list of distinct integer identifiers of sequences in the collection that contain the k-mer at least once. This is usually implemented as a dictionary, where the keys are the indexed k-mers and the values are *color set ids* that point to the color set associated with the k-mer. In the language of information retrieval, this general structure resembles an *inverted index* [31, 42], long the workhorse of web search engines, where the terms are k-mers and the postings (document) lists are color sets. This structure was rediscovered in the bioinformatics community over a decade ago as the colored de Bruijn graph [17], and has been much developed since. A survey of many methods can be found in Marchet et al. [26, 27]. Figure 2(a) illustrates a k-mer index.

Pseudoalignment tools differ primarily in their choice of data structure for representing the m distinct k-mers in the indeded collection and their associated d color sets. Themisto [5] represents the k-mer dictionary with the spectral Burrows-Wheeler transform (SBWT) and

the color sets as either arrays of 32-bit integers (for sparse sets) or as bit vectors (for dense sets). Fulgor [10] uses a minimal perfect hash function [29] for the k-mer dictionary and has an range of options for the color sets: mFulgor groups genomes into meta-colors removing repetitions across similar genomes, dFulgor encodes each color set relative to a chosen reference, and mdFulgor applies both compression layers together. Other tools aggregate k-mers and their relative color set using Bloom Filters organized either in a tree (Sequence Bloom Tree [39] and its derivatives [14, 40, 41]) or in a matrix structure (BGSI [7], COBS [6]).

Loosely speaking, the goal of pseudoalignment is to determine which colors (i.e. indexed sequence) match a given query, with a match determined as some function of shared features (traditionally, k-mers) between the query and the sequence. The k-mer based approaches mentioned above typically employ a threshold-based pseudoalignment strategy: all k-mers of a query sequence are looked up in the index, and a match with sequence S is reported if at least a fraction τ of the found k-mers occur in S. A typical value for τ is 0.8, and k is often 31 when dealing with bacterial genomes. More formally, given a set of documents (e.g. reference genomes) $D_1, D_2, \ldots, D_m$, a query Q, and $\tau \in 0..1$, we say Q *matches* or *pseudoaligns* to document D_i if the number of k-mers Q that occur in D_i exceeds τ of the total number of k-mers of Q that occur in the set of documents.

k-mer dict.	$\mathcal{C}_k$
AAGT	{●}
ACAG	{●, ●}
AGGA	{●}
...	...
TAAG	{●, ●}
TAGG	{●, ●}

(a) k-mer index.

finimizer dict.	$\mathcal{C}_f$
AAG	{●}
CA	{●, ●}
GA	{●}
GGT	{●, ●, ●}
TC	{●}
TA	{●, ●, ●}

(b) Finimizer index.

minimizer dict.	$\mathcal{C}_m$
AC	{●, ●}
AG	{●, ●, ●}
GA	{●}
GG	{●, ●, ●}
GT	{●, ●}

(c) Minimizer index.

Figure 2 Pseudoalignment indexes. In general, a pseudoalignment index for a collection of sequences can be viewed as consisting of a dictionary of features and, for each distinct feature, a set of colors (sequence ids) in which the feature occurs. Three pseudoalignment indexes for the set of genomes $\mathcal{T} = \{$ACAGGTAAGTC, ACAGGTAGGTAAG, AGGTAGGAAA$\}$ are shown above: (a) the k-mer-based index with $k = 4$; (b) the finimizer index; (c) the minimizer index with $m = 2$. It is instructive to compare these to the de Bruijn graph pictures in Fig. 1. To pseudoalign a read to an indexed collection, index features appearing in the read are first computed and then the color sets corresponding to these features are aggregated according to a pseudoalignment criterion.

The dictionary/color sets structure followed by k-mer indexes generalizes naturally to features other than k-mers. The dictionary stores the features in a data structure that supports rapid lookup, and the color sets contain the ids of sequences in the collection that contain those features. Queries are resolved in the same way as for k-mers, though the results may now, of course, differ. In Kaminari [21] and Raptor [37] the dictionary contains minimizers and the color set associated with a given minimizer x is the union of the color sets of the k-mers in the collection for which x is the minimizer. Kaminari represents the minimizer dictionary as a minimal perfect hash function [29] and the color sets in the same representation that Fulgor [10], mentioned above, uses for k-mer color sets. Raptor uses a kind of Bloom filter [28] of the minimizers for its dictionary, and may thus return errors at lookup time. The color sets are represented simply as 32-bit integer lists.

These "approximate" pseudoalignment schemes introduce an additional level of approximation on top of what k-mer based pseudoalignment has relative to full sequence alignment. The aim is to improve scalability of indexing while keeping false positives acceptably low.

The result of approximate pseudoalignment is indeed a superset of the true set of documents in which the k-mers of Q appear $\mathcal{C}_k$; the challenge is that of making it as close as possible to $\mathcal{C}_k$, while still enabling fast and memory-efficient queries. See Figure 2(b) and Figure 2(c).

In the next section we describe Finimap, a new pseudoalignment index that uses finimizers as features instead of k-mers or minimizers.

4 Finimap

In this section we describe Finimap, an alternative approach to pseudoalignment based on *finimizers* – shortest unique substrings in the de Bruijn graph, defined earlier. Finimap uses finimizers as features for approximate pseudoalignment, and as an alternative to k-mers, which are used by established pseudoalignment tools [5, 12, 15], and minimizers, which have been used recently in some tools [21, 37].

In Finimap, each k-mer is associated with a finimizer. Finimizers have three properties that maintain the local structure of the de Bruijn graph:

1. Consecutive k-mers tend to share the same finimizer: As a result the number of distinct finimizers is drastically smaller than the number of k-mers, enabling a substantial reduction of the index size.
2. Consecutive k-mers tend to share the same or very similar color sets: Together with property 1, this implies that when a finimizer is observed, there is a high chance of observing its associated k-mers. This helps to limit the number of false positives in $\mathcal{C}_f(x)$ for all k-mers x, since $\mathcal{C}_f(x)$ is formed from the union of similar sets. And it is in contrast with previously discussed approaches of combining color sets of randomly-selected k-mers, e.g. using Bloom filters, which might have substantially different color sets.
3. Disjoint k-mers in the de Bruijn graph – i.e. nodes that are not connected – cannot share the same shortest unique finimizer. Such unconnected nodes do not belong to the same sequence and therefore do not share the same color set. Consequently, finimizers color sets are more similar to those of single k-mers.

Figure 1 (top) illustrates the colored de Bruijn graph of three genomes together with the partitioning of that graph induced by finimizers (bottom left) and minimizers (bottom right). The figure clearly shows that the two properties explained above are respected. It is also apparent that Property 3 is not respected by minimizers, as k-mers not connected in the graph share the same minimizer, and are in the same partition. The practical implications of this will be discussed later.

The whole Finimap index includes: finimizers each associated to a color set id, and color sets – both compressed as explained in the next sections.

4.1 Color set representation

Because two or more finimizers could share the same color set (e.g. AAG and TC in Figure 2b), the color sets are deduplicated and each finimizer is assigned a unique color set identifier, that is the index of the color set in the color set representation – described below.

Let C be the total number of colors in the dataset, rows in the matrix, the deduplicated color sets are divided into three classes based on the color set size:

- *Sparse color sets*: A set is classified as sparse if the number of colors in the set is less than 1/16 of all the colors. Present colors are stored a integers and listed one after the other. Ending positions of each set are also stored.

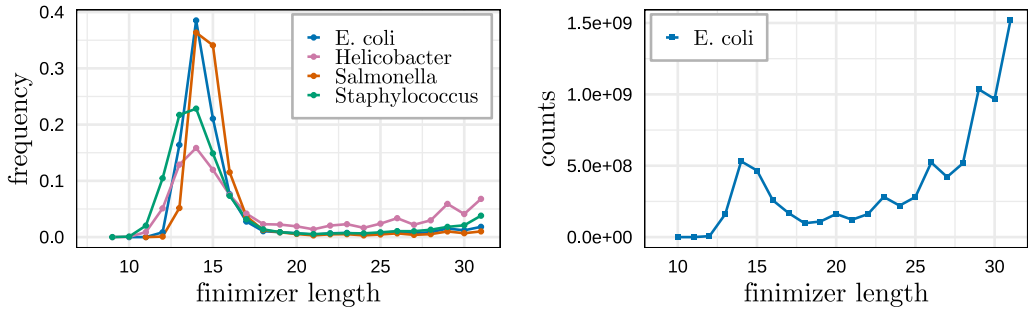


Figure 3 (Left) Length distribution of the distinct shortest unique finimizers with $k = 31$. The frequency on the y-axis is normalized by the total counts of finimizers for each dataset.
(Right) Distribution of finimizer lengths, one finimizer per every 31-mer, in the E. coli index.

- *Dense color sets*: If the set size is in between $[C/16, 15C/16]$, then the set is encoded though a bitvector of C bits. Bits are set only for colors present in the set.
- *Very dense sets*: If the number of colors in the set is higher than $15C/16$, we encode only absent colors using the same encoding used for sparse sets.

The sparse and very dense set are concatenated, and ending positions of each of these sets, monotonically increasing integer sequences, are stored using delta encoding combined with periodic checkpoints sampled every 32 elements. This enables compression while still allowing for constant-time random access. These techniques are inspired by Fulgor [10] and Themisto [5], while being different from both. Figure 9, in Appendix B illustrates the partitioning of the distinct color sets into these three classes across the datasets used in our experiments.

4.2 Finimizer dictionary structure

Finimap uses an ad hoc data structure to store and retrieve the finimizers of the indexed collection, which we outline in this section. Our aim was for a dictionary data structure that was fast and that used a reasonable amount of space. We emphasize that this data structure is only a proof-of-concept and is improvable in several ways, both in lookup time and space usage. Future releases of Finimap will explore these improvements, and we sketch what some of them might be throughout this section. Having said this, the current unoptimized version is already pleasingly fast, as our experiments in Section 5 show.

Finimizer Lookup. The key operation supported by the dictionary structure when processing a query read Q is to compute, for each $i \in 1..|Q|$, the prefix of $Q[i..|Q|]$, that is a shortest unique finimizer in the dictionary, or, if there is no finimizer, report $\emptyset$.

The dictionary is composed of two parts: A, which stores all finimizers having length less than or equal to a user specified parameter p, and B which stores the remaining longer finimizers. Let n_s denote the number of short finimizers (i.e. those with length less than or equal to p) and n_g denote the number of long finimizers. For ease of exposition we assume the k-mer length $k \leq 32$. Recall that k is also an upperbound on the length of any finimizer.

The rationale for splitting the finimizers into short and long sets this way is the observed finimizer length distribution shown in Figure 3 (see also the original paper on finimizers [2]). The division into two sets is a somewhat crude means of allowing us to represent finimizers shorter than p using less bits. A more nuanced approach will be explored in future work.

Structure for Short Finimizers. A is essentially an Elias–Fano-like data structure [11, 13] with bucket size $b = 256$, which we now describe. We interpret the short finimizers (i.e., those which are to be stored in A) as integers of $2p$ bits. In particular, each short finimizer is padded with 'A' symbols (the symbols are appended) until it is of length p. Every symbol is then mapped to a 2-bit integer in the natural way: A $\to$ 00, C $\to$ 01, G $\to$ 10, T $\to$ 11.

For example, with $p = 12$ the finimizer GTACCT of length 6 would become the integer 101100010111000000000000. With $\mathrm{enc}_p(T)$ we denote the binary string of length $2p$ (integer) that results from applying this encoding to the nucleotide string T. With $\mathrm{lcp}(i, j)$ we denote the length in bits of the longest common prefix of binary encodings of integers i and j.

After encoding the short finimizers as described above, we have a set S_A of n_s integers $a_0 < a_1 < \ldots < a_{n_s-1}$ over a universe U, where $|U| = u = 4^p$ – DNA alphabet is of size 4. Observe that the order of the finimizer encodings in S_A corresponds to the lexicographic ordering of the short finimizers, since A<C<G<T corresponds to $00 < 01 < 10 < 11$. Let $E_s = \ell_0, \ell_1, \ldots, \ell_{n_s-1}$ be the sequence of finimizer lengths corresponding to the short finimizers in the order in which their encodings appear in S_A. Our lookup procedure for finimizers makes use of the following easily proved lemma.

▶ **Lemma 6.** *Let $x = \mathrm{enc}_p(Q[i..i + k])$ be the integer encoding of the k-mer starting at position i in Q and let $a_j = \mathrm{pred}(x, S_A)$ be the predecessor of x in S_A. The shortest unique finimizer that is a prefix of $Q[i..i + k]$ is a_j iff $\mathrm{lcp}(x, a_j)/2 \geq \ell_j$.*

The lemma allows us to reduce prefix search on the set of finimizer strings to predecessor search on the set of finimizer encodings. The key observation is that if a short finimizer of length ℓ is a prefix of a query string, then their binary encodings share a common prefix of length at least 2ℓ. Hence, after locating the predecessor encoding, we can determine whether it corresponds to a valid finimizer prefix by checking the length of the common prefix.

We will now describe the Elias–Fano-like data structure A, used for storing S_A, and describe how it supports predecessor queries on S_A. A toy example of this is shown in Figure 4.

short finimizers	padded finimizers	integer ecodings (S_A)	bucket	lower 8 bits (L)
ACGG	ACGGAA	$000110100000 = 416$	1	160
ACGT	ACGTAA	$000110110000 = 432$	1	176
AGA	AGAAAA	$001000000000 = 512$	2	0
CTGAAA	CTGAAA	$011110000000 = 1920$	7	128

Figure 4 Example of the data structure A used to store short finimizers ($p = 6$), and support predecessor queries. $E = [4, 4, 3, 6]$, $L = [160, 176, 0, 128]$, $B = [0, 1, 1, 0, 0, 0, 0, 1]$, $H = [-, 0, 2, 2, 2, 2, 2, 3]$, buckets 1 start at $L[0]$, bucket 2 starts at $L[2]$, bucket 7 starts at $L[3]$. Note that B and H are not stored separately.

To support efficient predecessor queries, we partition each encoded integer into two parts: its lower 8 bits and its remaining higher bits. The higher bits determine the bucket of the integer (finimizer), while the lower 8 bits are stored explicitly in L. Let L be an array of n_s bytes where $L[x]$ is the lower 8 bits of $a_i \in S$. We conceptually divide the universe into buckets of size 256, where the jth bucket, denoted b_j, corresponds to the interval $[256j, 256(j + 1))$. All integers within the same bucket share the same higher bits and differ only in their lower 8 bits. We store a bitvector B such at $B[j] = 1$ if and only if there is at least one element from S_A falling in bucket b_j. We also store an array H of $u/256$ integers, one for each bucket, which contains indexes into L, defined as follows. Assume one or more elements from S_A fall into bucket b_j (i.e., $B[j] = 1$) and let $a_i \in S$ be the smallest of those

elements. We set $H[j] = x$ indicating that $L[x]$ is the first entry in L containing lower bits corresponding to elements of S_A falling in bucket b_j. If, on the other hand, b_j is empty, then we set $H[j] = y$, such that a_y is the predecessor of $256j$ in S_A; in other words, a_y is the largest element in S_A that lies before the start of bucket b_j.

To answer query $\text{pred}(x, S_A)$, we let $j = \lfloor x/256 \rfloor$ be the index of the bucket containing x. If the bucket containing x, i.e., b_j, is empty (determined by checking $B[j]$) then the answer to $\text{pred}(x, S_A)$ is given by $H[j]$. Otherwise, if i_j's bucket is nonempty, we calculate the number of elements in it as $c = H[j+1] - H[j] + B[j+1]$ and we scan $L[H[j]..H[j] + c - 1]$ until we determine the predecessor of x, or, if x is smaller than all elements in the bucket we return $H[j] - 1$ as the index of the predecessor. The scan is then limited to the bucket size, making it cache friendly in practice.

When implementing the above scheme we use an array of bytes for L and array of 32-bit integers for H. We use the most significant bit of $H[i]$ to store the bit $B[i]$. Queries are very simple, involving just a single access to H followed by a possible scan of at most 256 bytes. The space used is $u/8 + 8n_s$ bits. The array E of finimizer lengths, needed to apply Lemma 6, is stored as a separate array of bytes.

Looking at the example in Figure 4, and given a query 12-mer[2] $Q =$ACGTTAGTGTCA, $\text{enc}_6(Q) = 000110111100 = 444$. We need to find $\text{pred}(444, S_A)$. First we check the bucket: $\lfloor 444/256 \rfloor = 1$. Bucket 1 is not empty ($B[1] = 1$), $H[1] = 0$, $L[0] = 160$ and $\text{lcp}(444, 160)/2 < E[0]$, $3 < 4$, so since $444 > 416$, we need to check the next value in bucket 1. $L[1] = 176$ and $\text{lcp}(444, 176)/2 = E[1] = 4$, thus ACGT is a valid finimizer for Q.

There are many ways in which the above structure could be improved. The bucket size of $b = 256$ was chosen the ease implementation as it allows to sequences of lower bits in the EF structure to be stored as an array of bytes. It is almost certainly not the optimal setting for the EF bucket size, which would be $4^p/n_s$. Future work will explore this setting. Also, the array H can be replaced with a bit vector of the bucket sizes encoded in unary, as with usual EF implementations – see, e.g., [23]. Storing E as bytes is also excessive (for $p \leq 16$ 4 bits per entry is sufficient). Also, E could be interleaved with L to reduce cache misses.

Structure for Long Finimizers. Let S_B denote the lexicographically ordered set of long finimizers, those with length greater than p. The structure for long finimizers, B, is essentially a sorted array of 64-bit integers. As with the short finimizers above, we interpret long finimizers as integers. In particular, long finimizers are interpreted as 64-bit integers (recall that we are assuming a maximum k, and hence maximum finimizer length, of 32) after padding them to be of length 32 by appending 'A' symbols. In a slight abuse of notation, we will also use S_B to refer to this sorted array of finimizers encoded as integers. We also store an array L_B of finimizer lengths, similar to L in the A structure above.

The same observation about prefix search and predecessor queries used above (i.e., Lemma 6) for short finimizers applies here. That is, to find the shortest long finimizer that is a prefix of $Q[i..i + k]$, we only need to find the predecessor i_j of $Q[i..i + k]$ in S_B and then check whether the longest common prefix of that element and $Q[i..i + k]$ is at least ℓ_j. Because we store the contents of S_B sorted, answering predecessor queries is easy with binary search. We speed this process up by storing a lookup table of 4^q entries one for each possible prefix of length q. The ith entry of the lookup table contains the subarray of S_B containing all long finimizers prefixed with the ith prefix of length q and allows us to jump to the subarray containing finimizers that share that prefix, thus skipping some rounds of binary search.

[2] $k = 12$ and $p = 6$ are just an example.

Much could be done to reduce space in this data structure too. For example, the prefix table removes the need to use to encode the first q symbols of each long finimizer. We also remark that the set of (shortest unique) finimizers is prefix free, and that our current dictionary lookup procedure does not exploit this, for example, to search finimizers in order of length, and terminate search early. We leave these and other optimizations as future work.

4.3 Index Construction

The Finimap index is built in two distinct phases: extraction and compression. In the first phase, the finimizers of the dataset are identified and their color sets are constructed. Then finimizers and color sets are compressed, as described in Section 4.2 and Section 4.1, respectively, and each finimizer is assigned a unique identifier corresponding to one of the compressed deduplicated color sets.

Finimizer identification. First, we build the SBWT index [4] of the k-mers of the input database, and the associated longest common suffix array [1]. Next, we run the streaming finimizer lookup algorithm described in [2] for each sequence in the input database, against the SBWT index of the database. This provides us with the finimizer of each k-mer, represented as a pair (ℓ, r), where ℓ is the length of the finimizer and r is the colexicographic rank of the unique k-mer that has the finimizer ℓ-mer as a suffix. We store the finimizer lengths in an array $F[0..n)$, where n is the length of the SBWT, such that if k-mer with colexicographic rank r is suffixed by a finimizer, then $F[r]$ is the length ℓ of the finimizer, and otherwise $F[r] = 0$. Computing F is done in parallel with atomic updates on F.

Color set construction. We construct a bit matrix encoding the color sets of all the finimizers. This is a temporary representation of the color set information, which is eventually deduplicated and stored in the compressed form described above. Let N be the number of identified finimizers, and C be the number of colors. We build a matrix $M[0..N-1][0..C-1]$, such that $M[i][j] = 1$ iff the finimizer with rank i among all the identified finimizers has color j. We populate M by querying all input sequences again in parallel against the SBWT index, keeping track of the color of the current input sequence. We find the rank of the finimizer at colexicographic rank r via a rank query on a bit vector that marks the colexicographic ranks of the k-mers that are suffixed by finimizers. Color sets are then deduplicated and compressed as explained in Section 4.1.

This color set construction phase could be made more efficient by using Alanko and Puglisi's recent color set construction approach [3] to avoid realising the whole binary matrix.

4.4 Pseudoalignment Queries

Given a query read Q, Finimap processes each k-mer of Q by querying the finimizer dictionary structure to identify matching finimizers and retrieve their associated color sets. These retrieved color sets are then aggregated according to one of the two pseudoalignment criteria described below: *full-intersection* or *threshold-union*. The resulting aggregated color set constitutes the pseudoalignment result for Q. Finimap outputs a sorted list of results of the form $(genome, \#\text{found } k\text{-mers})$, ordered by the number of found k-mers in decreasing order.

Finimap supports two kinds of queries: *full-intersection* queries and *threshold-union* queries. Given x as a k-mer of Q we can define both.

Full intersection. A color c is included in the resulting set R if c is present in all the k-mers of Q found in the index; that is: $R = \bigcap_{x \in Q} \mathcal{C}_f(x)$.

Threshold union. Given a parameter $\tau \in [0, 1]$, a color c is included in the resulting set R if c is present in at least $t = \tau \times |Q|$, where $|Q|$ is the number of the k-mers of Q found in the index; that is: $R = \{c \mid |\{x \in Q : c \in \mathcal{C}_f(x)\}| \geq t\}$. Finimap outputs a list of results, sorted by the number of matches.

With $\tau = 1$, full intersection and threshold union queries results are the same. A common value for τ is 0.8, meaning that at least 80% of the found k-mers in $|Q|$ must be present in color c for it to be part of the final result. This threshold is adopted in experiments in Section 5. Figure 5 shows an example of a threshold union query with: color sets of k-mers $\mathcal{C}_k$, finimizers $\mathcal{C}_f$, and minimizers $\mathcal{C}_m$, and $\tau = 0.8$.

An important speed up in query time is given by dealing with dense color sets representations by considering only the absent colors and by lowering the number of resulting matches for absent colors and the final threshold $t = \tau \times |Q|$, subtracting the number of dense color sets encountered. This trick was described by the authors of Fulgor [10].

Q : AGTAAGTC	C_k		$\mathcal{C}_f$		$\mathcal{C}_m$
AGTA	$\emptyset$	AGTA	{●, ●, ●} ✗	AGTA	{●, ●, ●} ✗
GTAA	{●, ●}	GTAA	{●, ●, ●} ✗	GTAA	{●, ●}
TAAG	{●, ●}	TAAG	{●, ●, ●} ✗	TAAG	{●, ●, ●} ✗
AAGT	{●}	AAGT	{●}	AAGT	{●, ●, ●} ✗
AGTC	{●}	AGTC	{●}	AGTC	{●, ●, ●} ✗
	{● : 4, ● : 2}		{● : 5, ● : 3, ● : 3}		{● : 5, ● : 5, ● : 4}
result ($\tau = 0.8$):	{● : 4}		{● : 5}		{● : 5, ● : 5, ● : 4}

Figure 5 Example of a threshold union query, on our running example $\mathcal{T}$, using color sets of k-mers $\mathcal{C}_k$, fininimizers $\mathcal{C}_f$ (Finimap), and minimizers $\mathcal{C}_m$.

False positives. We compared the results of threshold union queries produced by Finimap (set $\mathcal{A}$, approximate, predicted by our tool) against the results obtained using exact k-mer-based pseudoalignment methods (set $\mathcal{X}$, exact), and defined the confusion matrix with respect to the exact pseudoalignment output. True positives (TP) are genomes present in both $\mathcal{A}$ and $\mathcal{X}$, whereas false positives are genomes present only in $\mathcal{A}$, i.e. genomes predicted by Finimap but not reported by the k-mer-based pseudoalignment methods.

In Finimap there are two main sources of false positives:

1. The finimizer color set is a superset of color sets of k-mers.
2. Alien k-mers that are not fully present in the index, but contain a finimizer in the index.

This second source is essentially an exacerbation of the first, arising when a non-indexed k-mer inherits the color set of one of the finimizers it contains.

As detailed in the next section, Finimap has fewer false positives than other approximate pseudoalignment tools, as it does not use Bloom filters or hash functions. Like other approximate methods, Finimap presents no false negatives as every k-mer in the index is associated to a finimizer.

An example of the result of a threshold union query can be found in Figure 5, where the color sets based on finimizers, $\mathcal{C}_f$, of GTAA and TAAG, differ from the single k-mers color sets, introducing false positives, due to $\mathcal{C}_f$ being a superset of $\mathcal{C}_k$ (source 1). AGTA is an alien k-mer not considered as such as it contains the finimizer TA (source 2). As shown in the example, false positives are introduced also by approximate methods using minimizers.

Table 1 Some approximate statistics on the datasets used for experiments. Distinct 31-mers include reverse complement.

	Ecoli	Salmonella	Staphylococcus	Helicobacter
Colors (genomes)	3,682	10,000	10,000	2,015
Distinct 31-mers ($\times 10^6$)	341	634	99	185
Distinct finimizers ($\times 10^6$)	50	92	16	27
Avg. finimizers length	15.73	15.58	16.05	19.15
Distinct color sets ($\times 10^6$)	4.82	4.56	2.04	6.73
Avg. genome length (Mb)	5.15	4.85	2.80	1.63

5 Experiments

We have implemented our tool FINIMAP in C++. In this section, we report experiments comparing its performance to other pseudoalignment tools. We tested threshold union queries with $k = 31$ and $\tau = 0.8$, on 50,000 positive queries of length 1000.

We compared FINIMAP against two so-called "approximate" pseudoalignment method indexes based on minimizers, KAMINARI [21] and RAPTOR [37], and also to the leading k-mer based pseudoalignment index FULGOR [10], which we use as a baseline. Default parameters were employed unless otherwise stated. Only a single thread of execution was used by all tools. See Appendix A for more details.

Experimental machine. Experiments were run on a dedicated Linux server running Ubuntu 20.04 with kernel 5.15.0-153-generic. The machine provides 504 GiB of available RAM, and is powered by an AMD Ryzen Threadripper PRO 3975WX processor (32 cores, up to 64 threads, 3.5 GHz clock, 32 × 32 KiB L1 cache, 32 × 512 KiB L2 cache, 128MiB L3 cache). Storage is provided by four 4 TB Seagate ST4000NM000A CMR HDDs, achieving speed of approximately 400 MiB/s, as measured using the `dd` utility.[3]

Datasets. In our experiments we consider four **bacterial** datasets that differ by number of genomes, average genome lengths, and internal similarity (repetitiveness).

These included: **E. coli**, a pangenome of 3682 *E. coli* whole genome sequences[4]; **Salmonella**, a collection of 10,000 *S. enterica* genomes[5]; **Staphylococcus**, the first 10,000 *S. aureus* genomes from AllTheBacteria dataset[6] [16]; **Helicobacter**, all 2,015 *H. pylori* whole-genomes in the AllTheBacteria dataset. Some statistics on these data sets are presented in Table 1.

Results. We compared the false positive rate (FPR = $\frac{\text{FP}}{(\text{FP}+\text{TN})}$) and query time of our tool to those of other approximate tools for pseudoalignment, and to Fulgor as a baseline – i.e. TP are colors reported by both the approximate tool and Fulgor and FP are colors only reported by the approximate tool.

False-positive rates are shown in Table 2 and the tradeoff between query time and false positive rate for each of the data sets is depicted in Figure 6. FINIMAP is always much faster and more accurate than KAMINARI. When KAMINARI is at its most accurate setting it

[3] Measured using the command `dd if=/dev/zero of=out.bin bs=1MB count=100000 oflag=direct status=progress`.
[4] The collection is available at `zenodo.org/record/6577997`.
[5] First 10,000 files from `http://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k/`.
[6] Available at `https://allthebacteria.org`.

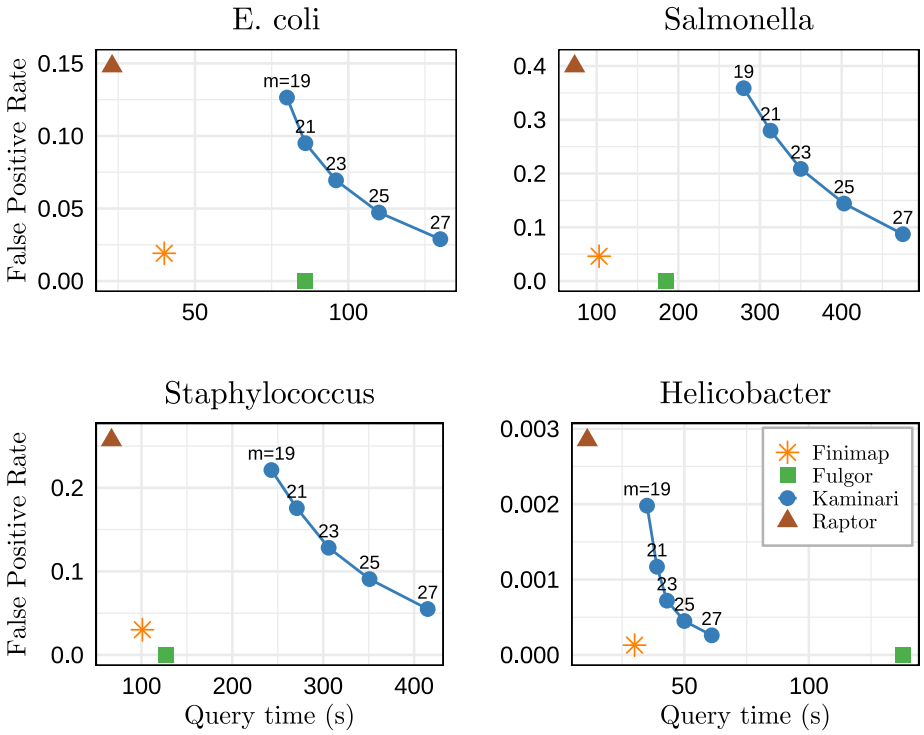


Figure 6 Results on threshold union queries with 50,000 positive queries of length 1,000, using 1 thread, $k = 31$ and $\tau = 0.8$. Fulgor represents the fastest version of the tool for each dataset: mdFulgor, Fulgor, dFulgor and mdFulgor. Note that Fulgor does not output a sorted list of colors and does not report the number of matches.

produces more false positives than Finimap, but is 2-5× slower. Finimap has comparable speed to Raptor and is always much more accurate, producing roughly a tenth of the false positives that Raptor does. In addition, Finimap is faster than Fulgor, the fastest k-mer based pseudoalignment tool. Full results on query time are in the Appendix B, Table 4.

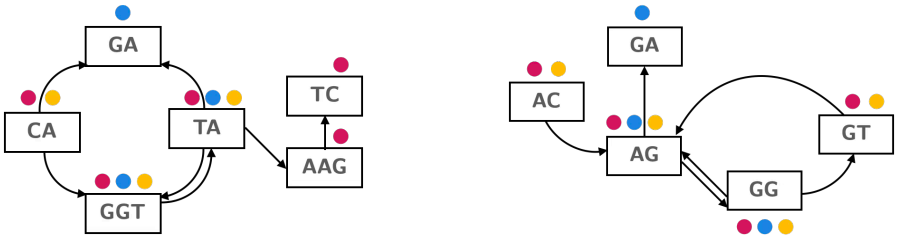
An important performance dimension that is not shown in the plots is that of index size. While it always fits comfortably in RAM, Finimap is always one of the larger indexes tested, about 3-6× bigger than Kaminari at its most accurate setting, with the lookup structure dominating. Finimap is smaller than Raptor, however. Full space results are in Table 3 (Appendix B). We are aware of several places in which the space usage of the current Finimap prototype can be reduced, and leave optimizing index size as future work.

6 k-merscolor sets vs. *inimizer color sets

In this section we compare k-mer color sets and color sets defined by finimizers and lexicographic minimizers. From our experimental results in the previous section, we observe that Finimap has a lower number of false positives compared to the minimizer-based tools Raptor and Kaminari (see Table 2). Here we explore in more detail the differences between the two general approaches, finimizers and minimizers, that could explain the observed variation in the false positive rate.

Table 2 False positive rate and percentage of true negatives of threshold union queries with 50,000 positive queries of length 1,000, $k = 31$ and $\tau = 0.8$. Color code is blue: $< 2 \times best$, orange: $< 4 \times best$, magenta otherwise, where *best* is the smallest value in every column.

		Ecoli	Salmonella	Staphylococcus	Helicobacter
avg. FPR	Finimap	0.02	0.05	0.03	0.0001
	Raptor 19	0.15	0.40	0.26	0.0029
	Kaminari 19	0.13	0.36	0.22	0.0020
	Kaminari 21	0.10	0.28	0.18	0.0012
	Kaminari 23	0.07	0.21	0.13	0.0007
	Kaminari 25	0.05	0.14	0.09	0.0005
	Kaminari 27	0.03	0.09	0.05	0.0003
avg. % TN		77	59	60	98



(a) Finimizer-partitioned colored dBG of $\mathcal{T}$. **(b)** Minimizer-partitioned colored dBG of $\mathcal{T}$.

Figure 7 Finimizer- ((a) $t = 1$) and minimizer-partitioned ((b) $m = 2$) dBG of $\mathcal{T}$. The full dBG of $\mathcal{T}$ is shown in Figure 1.

6.1 De Bruijn graph partitions

As illustrated in Figure 1b, consecutive k-mers tend to share the same finimizer (Property 1), so that only adjacent k-mers in the de Bruijn graph are grouped together (Property 3). This is consistent with the well-known observation that neighboring k-mers in the de Bruijn graph generally exhibit identical or highly similar color sets (Property 2).

In contrast, although consecutive k-mers tend to share the same minimizer, the partition that minimizers induce on the de Bruijn graph does *not* result in only neighboring k-mers sharing the same partition (see Figure 1c). As a result, distant regions of the graph may be included in the same partition, unlike shortest unique finimizers, which preserve locality. Consequently, structural relationships of the original graph are kept in the finimizer minor as only connected compnents are grouped in the same partition (Figure 7a). They are not maintained in the minimizer-partitioned colored de Bruijn graph (Figure 7b), which does not induce a minor: in this example, the edge connecting GT to AG did not exist in the original de Bruijn graph. This suggests that minimizer-based partitioning can distort the underlying graph topology, leading to an increased number of false positives.

6.2 Jaccard index

To assess whether the different ways in which the de Bruijn graph is partitioned, described above, have a real impact on the resulting index and queries, we calculated a measure of similarity between the k-mer color sets $\mathcal{C}_k$ ($k = 31$), and $\mathcal{C}_m$ or $\mathcal{C}_f$, the color sets of minimizers or finimizers, respectively. We used the Jaccard index [18]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

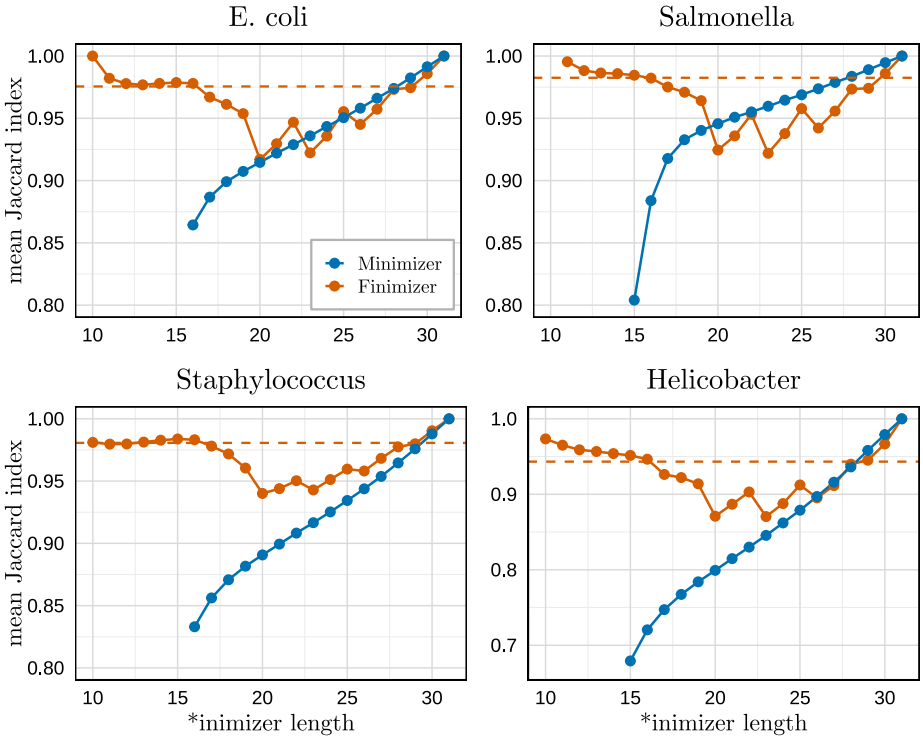


Figure 8 Plots of the average Jaccard index, the higher the better, of 31-mer color sets and minimizer color sets, in blue, and finimizer color sets, in orange. Since the length cannot be fixed for finimizers, the dashed line represents the weighted mean of the values for all finimizer color sets.

where A corresponds to $\mathcal{C}_k$, and B is either $\mathcal{C}_m$ or $\mathcal{C}_f$. This value was calculated considering all k-mer in the datasets with multiplicity 1 or higher. Note that in the formula, FN is always 0. Values range from 0 to 1, where 1 indicates perfectly identical sets, and 0 indicates that the sets are disjoint. Results on four different datasets are shown in Figure 8 for $k = 31$. We calculated the Jaccard index for every k-mer in the dataset; m from 15 to 31 for minimizers and $t = 1$ for finimizer. Since finimizers are not of fixed length, we calculated and plotted a weighted mean of the Jaccard index based on the frequency of each finimizer length.

On the E. coli data set with $m = 19$ (default parameter for Kaminari), the average Jaccard index of $\mathcal{C}_k$ and $\mathcal{C}_m$ is 0.9, but with finimizers it is 0.98. In order to achieve a similar value with minimizers we would require $m = 28$, which would clearly have a negative impact on query time, as shown by the trend plotted for Kaminari in Figure 6. For example, with E. coli the value of the mean Jaccard index is lower than the mean Jaccard index for every finimizer length. This suggests that on average, with $m = 19$, finimizer color sets $\mathcal{C}_f$ are more similar to $\mathcal{C}_k$ than minimizer color sets.

7 Conclusion and future work

We have described Finimap, a pseudoalignment index that uses finimizers, maximal unique substrings in the de Bruijn graph of the indexed data set, as features, rather than k-mers. Our experiments show that the results of Finimap are very similar to k-mer based pseudoalignment in that it produces far fewer false positives relative to k-mer based indexes

than pseudoalignment tools that use minimizers as features. We have focused on single-species bacterial genomics, which is a significant existing use case for pseudoalignment. Adapting and evaluating FINIMAP for pseudoalignment on broader, possibly more complex data sets, is an interesting direction for future work.

Our current implementation of FINIMAP is reasonably fast: it is always faster than the fastest k-mer based index FULGOR in our experiments, often significantly so. It also dominates the minimizer-based index KAMINARI, both in terms of speed and accuracy, and has comparable speed to RAPTOR while producing drastically fewer false positives. We believe our current implementation of FINIMAP can be further optimized, particularly in the lookup phase, in which an ad hoc two-part data structure is currently used. Trie data structures and hashtables stratified on key-length will be explored in future work. More generally, parallelism is a currently unexplored direction for optimization.

In addition, we remark that pseudoalignment indexes such as FINIMAP and KAMINARI get a head start on the k-mer based tools they try to emulate, like THEMISTO or FULGOR, by essentially storing some color sets pre-merged, so that less color sets are touched at query time. It may be interesting to try to quantify the computation saved this way and to try to arrive at upperbounds on the speed gains achievable by these new methods – ultimately, it may be possible to accelerate exact tools by caching partial results over a batch of queries.

Finally we are aware of the limitations of pseudoalignment methods that we used as ground truth for our experiments, and believe additional validation with simulated reads with a known ground truth or alignment-based methods such as Minimap2 [22] would provide stronger biological evidence for the accuracy of the proposed method.

References

1 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval*, LNCS 14240, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.

2 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Finimizers: Variable-length bounded-frequency minimizers for k-mer sets. *IEEE Transactions on Computational Biology and Bioinformatics*, 22(2):899–910, 2025. doi:10.1109/TCBBIO.2025.3545285.

3 Jarno N. Alanko and Simon J. Puglisi. Construction of distinct k-mer color sets via set fingerprinting. *bioRxiv*, 2026. doi:10.64898/2026.02.16.706153.

4 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *Proc. ACDA*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.

5 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Suppl 1):i260–i269, June 2023. doi:10.1093/bioinformatics/btad233.

6 Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. COBS: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019. doi:10.1007/978-3-030-32686-9_21.

7 Phelim Bradley, Henk den Bakker, Eduardo Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*, 37:152–159, February 2019. doi:10.1038/s41587-018-0010-1.

8 Nicolas Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature Biotechnology*, 34:525–527, April 2016. doi:10.1038/nbt.3519.

9 Alessio Campanelli, Giulio Ermanno Pibiri, Jason Fan, and Robert Patro. Where the patterns are: Repetition-aware compression for colored de bruijn graphs. *Journal of computational biology : a journal of computational molecular cell biology*, 31, October 2024. doi:10.1089/cmb.2024.0714.

10 Alessio Campanelli, Giulio Ermanno Pibiri, and Rob Patro. Fast Pseudoalignment Queries on Compressed Colored de Bruijn Graphs. In Broňa Brejová and Rob Patro, editors, *25th International Conference on Algorithms for Bioinformatics (WABI 2025)*, volume 344 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2025.6.

11 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974. doi:10.1145/321812.321820.

12 Jason Fan, Jamshed Khan, Noor Singh, Giulio Ermanno Pibiri, and Rob Patro. Fulgor: a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*, 19, January 2024. doi:10.1186/s13015-024-00251-9.

13 R.M. Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group. MIT, 1971. URL: https://books.google.fi/books?id=07DeGwAACAAJ.

14 Robert Harris and Paul Medvedev. Improved representation of sequence bloom trees. *Bioinformatics (Oxford, England)*, 36, August 2019. doi:10.1093/bioinformatics/btz662.

15 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome biology*, 21:249, September 2020. doi:10.1186/s13059-020-02135-8.

16 Martin Hunt, Leandro Lima, Daniel Anderson, George Bouras, Michael Hall, Jane Hawkey, Oliver Schwengers, Wei Shen, John A. Lees, and Zamin Iqbal. Allthebacteria – all bacterial genomes assembled, available, and searchable. *bioRxiv*, 2025. doi:10.1101/2024.03.08.584059.

17 Zamin Iqbal, Isaac Turner Mario Caccamo, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226–232, 2012. doi:10.1038/ng.1028.

18 Paul Jaccard. Etude de la distribution florale dans une portion des alpes et du jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:547–579, January 1901. doi:10.5169/seals-266450.

19 Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. Indexing all life's known biological sequences. *bioRxiv*, 2024. doi:10.1101/2020.10.01.322164.

20 Tamim Khawaja, Tommi Mäklin, Teemu Kallonen, Rebecca A. Gladstone, Anna K. Pöntinen, Sointu Mero, Harry A. Thorpe, Ørjan Samuelsen, Julian Parkhill, Mateen Izhar, M. Waheed Akhtar, Jukka Corander, and Anu Kantele. Deep sequencing of escherichia coli exposes colonisation diversity and impact of antibiotics in punjab, pakistan. *Nat. Commun.*, 15(5196), 2024. doi:10.1038/s41467-024-49591-5.

21 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. *Bioinformatics Advances*, 6(1):vbag120, January 2026. doi:10.1093/bioadv/vbag120.

22 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, September 2018. doi:10.1093/bioinformatics/bty191.

23 Danyang Ma, Simon J. Puglisi, Rajeev Raman, and Bella Zhukova. On Elias-Fano for rank queries in FM-indexes. In *Proc. 31st Data Compression Conference (DCC)*, pages 223–232. IEEE, 2021. doi:10.1109/DCC50243.2021.00030.

24 Tommi Mäklin, Teemu Kallonen, Jarno Alanko, Ørjan Samuelsen, Kristin Hegstad, Veli Mäkinen, Jukka Corander, Eva Heinz, and Antti Honkela. Bacterial genomic epidemiology with mixed samples. *Microbial genomics*, 7(11), 2021. doi:10.1099/mgen.0.000691.

25 Tommi Mäklin, Teemu Kallonen, Sophia David, Christine J Boinett, Ben Pascoe, Guillaume Méric, David M Aanensen, Edward J Feil, Stephen Baker, Julian Parkhill, et al. High-resolution sweep metagenomics using fast probabilistic inference. *Wellcome open research*, 5, 2020. doi:10.12688/wellcomeopenres.15639.2.

26 Camille Marchet. Advancements in colored k-mer sets: essentials for the curious. *arXiv preprint*, September 2024. doi:10.48550/arXiv.2409.05214.

27 Camille Marchet, Christina Boucher, Simon Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k -mers for querying large collections of sequencing data sets. *Genome Research*, 31, December 2020. doi:10.1101/gr.260604.119.

28 Svenja Mehringer, Enrico Seiler, Felix Droop, Mitra Darvish, René Rahn, Martin Vingron, and Reinert Knut. Hierarchical interleaved bloom filter: enabling ultrafast, approximate sequence queries. *Genome Biology*, 24, May 2023. doi:10.1186/s13059-023-02971-4.

29 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Suppl 1):i185–i194, June 2022. doi:10.1093/bioinformatics/btac245.

30 Giulio Ermanno Pibiri, Jason Fan, and Rob Patro. Meta-colored compacted de bruijn graphs. In Jian Ma, editor, *Research in Computational Molecular Biology*, pages 131–146. Springer Nature Switzerland, 2024. doi:10.1007/978-1-0716-3989-4_9.

31 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6), December 2020. doi:10.1145/3415148.

32 Mark Reppell and John Novembre. Using pseudoalignment and base quality to accurately quantify microbial community composition. *PLoS computational biology*, 14(4):e1006096, 2018. doi:10.1371/journal.pcbi.1006096.

33 Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. doi:10.1093/bioinformatics/bth408.

34 Michael Roberts, Brian R. Hunt, James A. Yorke, Randall A. Bolanos, and Arthur L. Delcher. A preprocessor for shotgun assembly of large genomes. *J. Computational Biology*, 11(4):734–752, 2004. doi:10.1089/cmb.2004.11.734.

35 Lorian Schaeffer, Harold Pimentel, Nicolas Bray, Páll Melsted, and Lior Pachter. Pseudoalignment for metagenomic read assignment. *Bioinformatics*, 33(14):2082–2088, 2017. doi:10.1093/bioinformatics/btx106.

36 Saul Schleimer, Daniel Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 10, April 2003. doi:10.1145/872757.872770.

37 Enrico Seiler, Svenja Mehringer, Mitra Darvish, Etienne Turc, and Reinert Knut. Raptor: A fast and space-efficient pre-filter for querying very large collections of nucleotide sequences. *iScience*, 24:102782, June 2021. doi:10.1016/j.isci.2021.102782.

38 Sergey Shiryev and Richa Agarwala. Indexing and searching petabase-scale nucleotide resources. *Nature Methods*, 21:1–9, May 2024. doi:10.1038/s41592-024-02280-z.

39 Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3):300, 2016. doi:10.1038/nbt.3442.

40 Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. In *21st Annual International Conference on Research in Computational Molecular Biology - Volume 10229*, RECOMB 2017, pages 257–271, Berlin, Heidelberg, 2017. Springer-Verlag. doi:10.1007/978-3-319-56970-3_16.

41 Sanjay Kumar Srikakulam, Sebastian Keller, Fawaz Dabbaghie, Robert Bals, and Olga Kalinina. Metaprofi: an ultrafast chunked bloom filter for storing and querying protein and nucleotide sequence data for accurate identification of functionally relevant genetic variants. *Bioinformatics*, 39, February 2023. doi:10.1093/bioinformatics/btad101.

42 Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006. doi:10.1145/1132956.1132959.

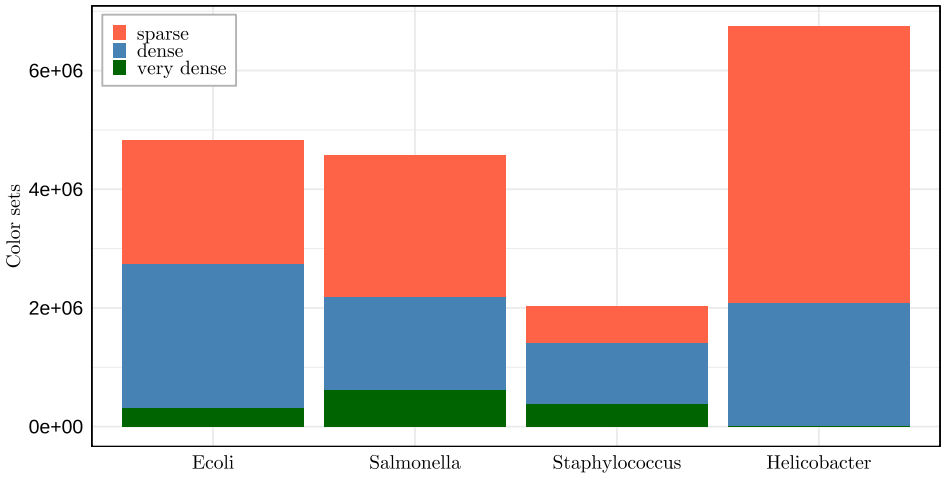


Figure 9 Finimap color sets representations in terms of number of color sets for each of the three representations.

A Tool versions and parameters used

- Finimap: parameters `build-fmin -k 31 -x 18 -p 10`
- Fulgor: commit `ef65419`, parameter `-k 31 -m 19`
- Kaminari: commit `0c403d1`, parameters `-k 31 -a -m 19,21,23,25,27`
- Raptor: commit `fa80e93`, parameters `-kmer 19 -window 31`

A threshold τ of 0.8 was used in all the tools for threshold union queries.

B Additional data

This section contains some additional results that where not included in the main body.

In Figure 9 it is shown for each dataset the partition of color sets in the three representations: sparse, dense and very dense. The dense representation is the most used in Ecoli and Staphylococcus, whereas in Salmonella and Helicobacter there are more sparse sets. In Helicobacter the sparse representation dominates the other two, and very few colorsets are stored as very dense.

Figure 10 shows FINIMAP index size split into colors and finimizers representations. Distinct dataset have different percentage of the index size dedicated to the two representations. For example, Helicobacter only contains 2015 genomes, but in spite of these genomes being the shortest out of the four used datasets, their high level of repetitiveness leads to 27 millions of finimizers quite long on average (19.15 bases) and this also creates a high number of distinct color sets. Nevertheless the size of color sets representations is smaller than that of other datasets due to the lower number of genomes; still the finimizers representation dominates. On the contrary, in Salmonella (10,000 genomes) the color sets representation dominates the index size with 4,564,828 distinct color sets. The color sets representation size of FINIMAP could be improved using techniques described in the last version of Fulgor [10].

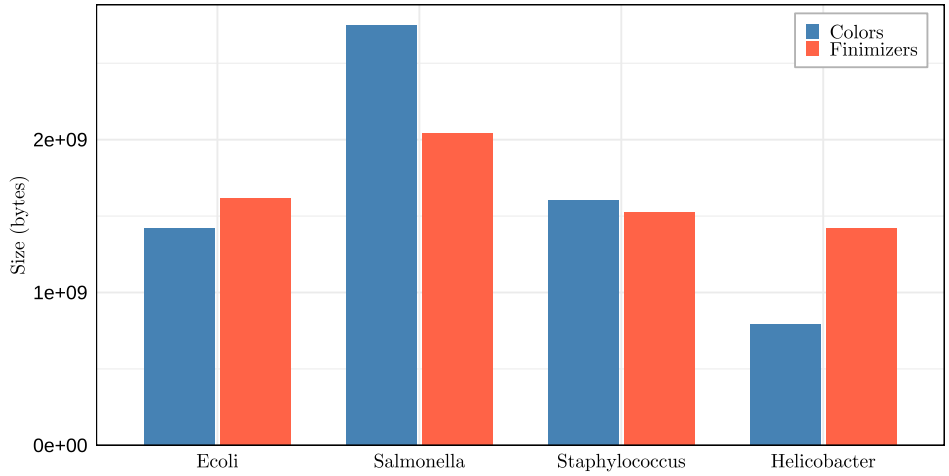


Figure 10 Finimap index size divided in colors and finimizers representations.

Table 3 Index size, in bits per k-mer, of all the tested tools with $k = 31$. blue: $< 2 \times best$, orange: $< 4 \times best$, magenta otherwise, where *best* is the smallest value in every column.

	Ecoli	Salmonella	Staphylococcus	Helicobacter
Finimap	71.20	60.32	230.24	95.68
Raptor 19	129.76	239.44	384.88	25.28
Kaminari 19	12.48	11.04	34.72	6.72
Kaminari 21	14.24	12.64	40.00	8.00
Kaminari 23	16.40	14.72	46.56	9.76
Kaminari 25	19.20	17.36	55.04	12.16
Kaminari 27	23.12	20.96	66.08	15.60
Fulgor	34.40	31.36	105.92	31.20
mFulgor	17.60	11.20	20.32	28.56
dFulgor	15.60	14.16	29.36	20.00
mdFulgor	12.56	9.20	16.48	22.16

Table 4 Query time in seconds of threshold union queries with 50,000 positive queries of length 1,000, 1 thread, $k = 31$ and $\tau = 0.8$. Color code is blue: $< 2 \times best$, orange: $< 4 \times best$, magenta otherwise, where *best* is the smallest value in every column.

	Ecoli	Salmonella	Staphylococcus	Helicobacter
Finimap	40	103	101	30
Raptor 19	23	73	67	11
Kaminari 19	80	280	243	34
Kaminari 21	86	313	271	39
Kaminari 23	96	350	306	43
Kaminari 25	110	403	351	50
Kaminari 27	130	475	415	61
Fulgor	99	185	187	157
mFulgor	111	194	140	180
dFulgor	111	260	80	180
mdFulgor	86	187	127	138