Taxonomic classification with maximal exact matches in KATKA kernels and minimizer digests

Dominika Draesslerová
Czech Technical University in Prague, Czech Republic

Omar Ahmed
Johns Hopkins University, USA

Travis Gagie
Dalhousie University, Canada

Jan Holub
Czech Technical University in Prague, Czech Republic

Ben Langmead
Johns Hopkins University, USA

Giovanni Manzini
University of Pisa, Italy

Gonzalo Navarro
University of Chile, Chile

Abstract

For taxonomic classification, we are asked to index the genomes in a phylogenetic tree such that later, given a DNA read, we can quickly choose a small subtree likely to contain the genome from which that read was drawn. Although popular classifiers such as Kraken use k-mers, recent research indicates that using maximal exact matches (MEMs) can lead to better classifications. For example, we can

- build an augmented FM-index over the the genomes in the tree concatenated in left-to-right order;
- for each MEM in a read, find the interval in the suffix array containing the starting positions of that MEM's occurrences in those genomes;
- find the minimum and maximum values stored in that interval;
- take the lowest common ancestor (LCA) of the genomes containing the characters at those positions.

This solution is practical, however, only when the total size of the genomes in the tree is fairly small. In this paper we consider applying the same solution to three lossily compressed representations of the genomes' concatenation:

- a KATKA kernel, which discards characters that are not in the first or last occurrence of any $k_{\max}$-tuple, for a parameter $k_{\max}$;
- a minimizer digest;
- a KATKA kernel of a minimizer digest.

With a test dataset and these three representations of it, simulated reads and various parameter settings, we checked how many reads' longest MEMs occurred only in the sequences from which those reads were generated ("true positive" reads). For some parameter settings we achieved significant compression while only slightly decreasing the true-positive rate.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Taxonomic classification, metagenomics, KATKA, maximal exact matches, string kernels, minimizer digests

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Kraken [28] is probably the best-known metagenomic tool for taxonomic classification. Given a phylogenetic tree for a collection of genomes and a value k, it stores an index mapping each k-mer in the collection to the root of the lowest subtree containing all occurrences of that k-mer. Later, given a DNA read — which may not match exactly in any of genomes in the collection — it tries to map all the k-mers in that read to subtrees in the tree and then to choose a small subtree likely to contain the source of the read. For example, if Kraken is given the toy phylogenetic tree shown at the top of Figure 1 and $k = 3$, then it will store the k-mer index shown at the bottom of that figure. Later, given the toy read `ATAC`, it will map `ATA` to 6 and `TAC` to 2. Since the subtree rooted at 6 contains the one rooted at 2, it will report that the read probably came from a genome in the subtree rooted at 2.

Nasko et al. [18] showed that a static choice of k is problematic, since "the [reference] database composition strongly influence[s] the performance", with larger k values working better as the collection of genomes grows over time. Limiting all analyses to a single choice of k causes other problems as well. First, some branches of the taxonomic tree are well studied and contain a large number of genome assemblies for diverse strains and species. Other branches are scientifically significant but harder to study, and contain only a few genomes. In the more richly sampled spaces, larger values of k will better allow for discrimination at deeper levels of the tree.

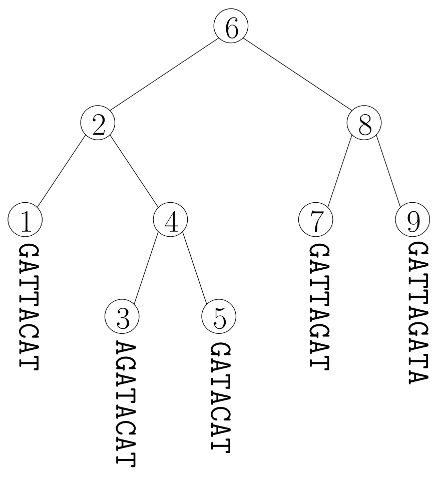
Choosing a constant value for k also conflicts with the varying error rates across sequencing technologies. For the high-accuracy Illumina technology, we expect longer matches to the data base and should favour a larger k. For a high-error-rate technology like Oxford Nanopore, we expect shorter matches and a small k is better. To this end, many widely tools for classifying long (error-prone) reads use matching statistics and/or full-text indexes [15, 1], as do some for short reads [14, 17]. Nasko et al. observed that

> "alternative approaches to traditional k-mer-based [lowest common ancestor] identification methods, such as those featured within KrakenHLL [4], Kallisto [3], and DUDes [21], will be required to maximize the benefit of longer reads coupled with ever-increasing reference sequence databases and improve sequence classification accuracy."

Cheng et al. [6] showed that finding the maximal exact matches (MEMs) of the read with respect to the collection and then mapping each MEM to the root of the lowest subtree containing all occurrences of that MEM, gives better results than mapping k-mers for any single k. However, they did not give a space- and time-efficient index for finding and mapping MEMs. As a potential step toward working with MEMs, Gagie et al. [10] described an LZ77-based index KATKA that takes $O(z \log n)$ space, where z is the number of phrases in the LZ77 parse of the collection of genomes and n is the total length of the collection, and works like Kraken but taking k at query time instead of at construction time.

KATKA finds the indices of the genomes containing the first and last occurrences of each k-mer in the collection, then performs a lowest common ancestor (LCA) query on those genomes in the tree to find the root of the smallest subtree containing all the occurrences of that k-mer. As far as we know, however, there is no practical way to find MEMs with LZ77- or grammar-based indexes, even if there have been some promising recent developments [12, 19] in this direction. Thus, KATKA is not yet a practical implementation of Cheng et al.'s idea.

Since an LCA data structure for the phylogenetic tree takes a constant number of bits per genome, the main challenge to implementing Cheng et al.'s idea is to find the MEMs of the read with respect to the collection and then to find the genomes containing the first and



ACA: 2	ATT: 6	TAC: 2
AGA: 6	CAT: 2	TAG: 8
ATA: 6	GAT: 6	TTA: 6

Figure 1 A toy phylogenetic tree **(top)** with Kraken's k-mer index for $k = 3$ **(bottom)**.

last occurrence of each MEM. We call all this information the *MEM table* for the read. We describe in Section 2 how we can extend a technique by Ohlebusch et al. [20] to build the MEM table in constant time per character in the read plus $O(\log n)$ time per MEM as long as we are willing to use an $O(n)$-bit augmented FM-index for the collection — but a space usage of $O(n)$ bits is prohibitive when the collection is large and anyway wasteful when it is highly repetitive. The most practical way we know of to build the MEM table is with Cáceres and Navarro's [5] block-tree compressed suffix tree, but that offers more functionality than we need at the cost of using more space than we would like ("1–3 bits per symbol in highly repetitive text collections").

In this paper we build approximations of MEM tables using augmented FM-indexes over

- a string kernel for the collection,
- a minimizer digest for the collection,
- a string kernel for a minimizer digest for the collection.

String kernels and minimizer digests are lossily compressed representations of strings, which we review in Section 2. We need a special kind of string kernel that we call a KATKA kernel and define also in Section 2. We can use KATKA kernels and minimizer digests to reduce the size of the augmented FM-index, at the cost of limiting the lengths of matches and reporting some false-positive matches. To test how we can trade off accuracy for compression, we built augmented FM-indexes over a test dataset and KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests for that dataset with various parameter settings, and checked for how many of a set of simulated reads their longest MEMs occurred only in the sequences from which those reads were generated ("true positive" reads). For some parameter settings we achieved significant compression while only slightly decreasing the true-positive rate.

2 Preliminaries

2.1 Augmented FM-indexes

Ohlebusch et al. [20] showed how, if we store an augmented FM-index, then when given a read we can find its MEMs quickly. We first show how to extend their technique to computing the MEM table in constant time per character in the read and $O(\log n)$ time per MEM.

Suppose each genome in the collection is terminated by a special separator character `$` as shown in Figure 2. The augmented FM-index consists of data structures supporting access, rank and select on the collection's Burrows-Wheeler Transform (BWT)[1]; access, range-minimum and range-maximum on their suffix array (SA); range-minimum, range-maximum, previous smaller value (PSV) and next smaller value (NSV) queries on their longest common prefix (LCP) array; and rank on the bitvector B with a 1 marking each `$` in the collection. As long as the collection is over a constant-size alphabet, these data structures together take $O(n)$ bits with all their queries taking at most $O(\log n)$ time. They are also implemented in the Succinct Data Structure Library (SDSL) [13] as components of a compressed suffix tree.

Given the read `ACATA`, for example, we start a backward search with BWT interval BWT[0..44] (the entire BWT). After 3 backward steps we find the interval BWT[16..18] for `ATA`. Since this interval does not contain a copy of the preceding character `C` in the read, we know `ATA` is a MEM of `ACATA` with respect to the collection. We use range-minimum and range-maximum queries over SA[16..18] and access to SA to determine that the first and last occurrences of `ATA` start at positions 11 and 41 in the collection. Since $B.\text{rank}_1(11) = 1$ and $B.\text{rank}_1(41) = 4$, we know those occurrences are in the second and fifth genomes in the collection (stored at nodes 3 and 9 in the phylogenetic tree). Notice we consider only the first and last occurrences and not the occurrence starting at position 19, for example.

We then use rank and select queries on the BWT to look for the previous copy BWT[14] of `C` and next copy of `C` (which does not exist); use a range-minimum query on LCP[$14+1 = 15$..16] to find the position 16 of the length 2 of the longest prefix `AT` of `ATA` that is preceded by `C` in the collection; use access to the LCP to retrieve that value 2; and use $\text{PSV}(16) = 12$ and $\text{NSV}(16) = 22$ queries to find the interval BWT[12..$22 - 1 = 21$] for that prefix `AT`. After 2 backward steps we find the interval BWT[6..8] for `ACAT`. We use range-minimum and range-maximum queries over SA[6..8] and access to SA to determine that the first and last occurrences of `ACAT` start at positions 4 and 21 in the collection. Since $B.\text{rank}_1(4) = 0$ and $B.\text{rank}_1(21) = 2$, we know those occurrences are in the first and third genomes in the collection (stored at nodes 1 and 5 in the phylogenetic tree).

2.2 String kernels and KATKA kernels

Ferrada, Gagie, Hirvola and Puglisi [8, 11] and Prochazka and Holub [22] (see also [9]) independently defined the order-$k_{\max}$ kernel of a string to be the subsequence consisting of the characters in the first occurrence of any distinct $k_{\max}$-mer in the string, with maximal omitted substrings replaced by copies of a new separator character `#`. Since we want to find the first and last occurrences of matches, we define the *order-$k_{\max}$ KATKA kernel* of a collection of genomes essentially the same way, but with the subsequence consisting of the characters in the first *or last* occurrence of any distinct $k_{\max}$-mer in the string, and the

[1] To reduce the size of the figure we have actually shown the genomes' extended BWT [16], which is functionally equivalent as far as we are concerned as long as each genomes has length $\Omega(\log n)$. Notice some LCP values, such as LCP[4], "wrap around" and count a character in the BWT.

i	SA[i]	LCP[i]	BWT[i]	context
0	17	0	T	`$AGATACA`
1	25	1	T	`$GATACA`
2	8	4	T	`$GATTACA`
3	34	6	T	`$GATTAGA`
4	44	9	A	`$GATTAGAT`
5	43	0	T	`A$GATTAGA`
6	13	1	T	`ACAT$AGA`
7	21	5	T	`ACAT$GA`
8	4	8	T	`ACAT$GAT`
9	30	1	T	`AGAT$GAT`
10	39	4	T	`AGATA$GAT`
11	9	5	$	`AGATACAT`
12	15	1	C	`AT$AGATA`
13	23	3	C	`AT$GATA`
14	6	6	C	`AT$GATTA`
15	32	8	G	`AT$GATTA`
16	41	2	G	`ATA$GATTA`
17	11	3	G	`ATACAT$A`
18	19	7	G	`ATACAT$`
19	1	2	G	`ATTACAT$`
20	27	4	G	`ATTAGAT$`
21	36	7	G	`ATTAGATA$`
22	14	0	A	`CAT$AGAT`
23	22	4	A	`CAT$GAT`
24	5	7	A	`CAT$GATT`
25	31	0	A	`GAT$GATT`
26	40	3	A	`GATA$GATT`
27	10	4	A	`GATACAT$`
28	18	8	$	`GATACAT`
29	0	3	$	`GATTACAT`
30	26	5	$	`GATTAGAT`
31	35	8	$	`GATTAGATA`
32	16	0	A	`T$AGATAC`
33	24	2	A	`T$GATAC`
34	7	5	A	`T$GATTAC`
35	33	7	A	`T$GATTAG`
36	42	1	A	`TA$GATTAG`
37	12	2	A	`TACAT$AG`
38	20	6	A	`TACAT$G`
39	3	8	T	`TACAT$GA`
40	29	2	T	`TAGAT$GA`
41	38	5	T	`TAGATA$GA`
42	2	1	A	`TTACAT$G`
43	28	3	A	`TTAGAT$G`
44	37	6	A	`TTAGATA$G`

G	A	T	T	A	C	A	T	$	A	G	A	T	A	C	A	T	$	G	A	T	A	C	A	T	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

G	A	T	T	A	G	A	T	$	G	A	T	T	A	G	A	T	A	$
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44

$$B = 000000001000000001000000010000000010000000001$$

Figure 2 The augmented FM-index for our toy collection of genomes.

copies of the separator character `$`. Since reads will not contain `$`, we also do not replace with `#` all maximal omitted substrings adjacent to copies of `$`.

By construction, for $k \leq k_{\max}$, every k-mer from the normal alphabet (so not including `$`) in the original string occurs in the KATKA kernel and vice versa. Moreover, if there are i copies of `$` to the left of the first occurrence of such a k-mer in the kernel, then the first occurrence of that k-mer in the collection is in the $(i+1)$st genome (and symmetrically for the last occurrences). The running example we have used so far is too small to illustrate properly the advantages and disadvantages of KATKA kernels, so Figure 3 shows a slightly larger collection of slightly longer toy genomes and Figure 4 shows the subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer and the copies of `$`. Figure 5 shows the 4th-order KATKA kernel of the collection with maximal omitted substrings replaced by copies of `#`. In this example, the 4th-order KATKA kernel is about half the size of the original collection, but this varies in practice depending on $k_{\max}$ and the size and repetitiveness of the collection. The 5th-order KATKA kernel, which we do not show, is about 70% of the size of the original collection.

```
ACTTAGCTGACGTTCCGGGTGTTTTTGGCCATCTTCTATAGATTTCCCAGAGACATACTAGGCGTGCTGAAGTTGTGACTCGCGGCCGTATTTTCTAACG$
ACTTAGCTGACGTTCCGGGTGTTTTAGGCCATCTTCTATAGATTTCTCAGAGACATAGTAGGCGTGCTGAAGTTGTGACTCGCGGCCGTATTCCCTAACG$
ACTTAGCTGACGTTCCGGGTGTTTTAGGCCATCTTCTATAGTTTTCTCAGAGACATACTAGGCGTGCTGAAGTTGTCACGCGCGCCCGTATTTCCTAACG$
ACTTAGCTGACGTTCAGGGTGTTTTAGGCCATCTTCTATAGTTTTCTCAGAGACATAGTAGGCGTGCTGAAGTTGTCACTCGCGCCCGTATTTCCTAACG$
TCAGAGCTGAGGTTCGGGGTGATTTAGGACATCTTCCATCGATTTCTCAGAGACGTCCTCAGCGTGCTCAAGTTGTCACCCGCGGCCGTTATTCCTAACG$
TCCGAGCTGAGGTTCGGGGTGGTTTAGGTCATCTTCTATAAATTTCTCAGAGACGTCCCCAGCGTGCTCAAGTTGTCACTCGCGGCCGTTTTTCCGAACG$
TCATAGCTGAGGTACGGGGTGGTTTAGGCCAGCTTCTATAGATTTCTCAGACACGAGCAGGGCGTGCTTAAGTTGTCACTCGCGGCCGTTTTTCCTAACG$
TCATAGCTGAGGTACGGGGTGGTTTAGGCCACCTTCTATAGATTTGTCAGACACGAGCTCGGCGTGCTGAAGTTGTCACCCGCGGCCGTTTTTCCTAACG$
TCCAAGCGTCCGTTCGGGGTGGGTTAGGCGATCTTCTGTAGAGTTCTCGGAGACAAGCTAGGCGTGCTGATGTTGTCATTCGCGGCCGTGTTCCCTAACG$
TCCAAGCTTCCGTTCGGGGTGGGTTAGACGATCTTCTGTACAGTTCTTTGAGACAAGCTAGGCGTGCTGAAGTTGTCACACGCGGCCGTGTTCCCTAACG$
TGACAGCGGACGTTCGGGGTGGGTTAGGACATCTTCCGTAGATTTCTCGGATACAAGCTAGGCGTTCTGAAGTTGGCACTCGCGGCCTTGTTCCCTAACG$
TCCCTGCTGACGATCGGGGTAGGTTAGGACATCTTCCGTTGATTTCTCGGATACAAGCTCGGCGTTCTGAAGTTGGCACTCGCGGCCGTGTTCCCTAACG$
TAATATCAGACGTTCGGGCTGGGCTAGTCCATCTTCTTTAGATTTCTCAGAGACTTGCTAGGCGTGCTGAAGTTGGCACTCGTGGCCGTGTTCCCTAACG$
TAATATCAGACGTTCGGGATGGGCTAGTCCATCTTCTTTAGATTTCTCAGAGACATGCTAGGCGTGCTGCAGTTGTCACTCGTGGCCGTGTTCCGTTACG$
TAATATCAGACGTTCGGGCTGGATTAGGCCATCTTCTTTAGATTTCTCAGAGACATGCTAGGCGTGCTGAAGTTGGTAATCGCGGCCCTGTTCTTTAACG$
TAATATCAGACGTTCGGGCCGGGTTAGGCCATCTTCTTTAGATTTCTCAGAGACATGCTAGGCGTGCTGAAGTTGGCAATCGCGGACCTGTTCTCTAACG$
```

Figure 3 A slightly larger collection of slightly longer toy genomes.

```
ACTTAGCTGACGTTCCGGGTGTTTTTGGCCATCTTCTATAGATTTCCCAGAGACATACTAGGCGTGCTGAAGTTGTGACTCGCGGCCGTATT  CTAACG$
                      TTTA                  TCTCAG     TAGTAG            TGTG            ATTCCCTA   $
                                                       TACTA             TGTCACGCGCGCCCG    TCCT    $
            TTCAGGG                                     AGTA                CACT GCGCC GTAT         $
   GAGCTGAGGTTCGGGGTGAT  AGGAC    TCCATCGAT          CGTCCTCAGCG GCTCAAG    CACCCGC    GTTATT       $
 CCGAG            GTGGT   GGTCAT     ATAAATT            CCCCA      TCAA                      CCGAAC $
          GGTACGG           CCAGCTT               ACACGAGCAGGGC   CTTAAG                            $
 CATAGC GAGG ACGG           CCACCTTCTATAG            CGAGC                  CACCCGC    GTTTTTCCT    $
 CCAAGCGTC         TGGG    GCGATC TCTGTAGAGT   CGGAGACAAGCTA        GATGT  TCATTC                   $
 CCAAGCTT                           TGTACAGT CTTTGAG                        CACACGC                 $
  ACAGCG         GGTG               CGTA        GGATA                     GGCAC     GCCTTG          $
  CCTGCTGACGATCGGGGTAGGT AGGA         TTGAT      GATACAAGCTC     TCTG                               $
TAATATCA        GGCTGG   AGTC                      GACTTGC                 GCACTCGT        TCCCTA   $
               GGGATGGG TAGTCCA                      CATGC        CTGCAGTTGTCACTCGTGG  GTGTTCCGTTACG$
                GGCTGGATTA                                               TGGTAATC CGGCCCT     TTAA  $
TAATATCAGACGTTCGGGCCGGGTTAGGCCATCTTCTTTAGATTTCTCAGAGACATGCTAGGCGTGCTGAAGTTGGCAATCGCGGACCTGTTCTCTAACG$
```

Figure 4 The subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer and the copies of $, with omitted characters replaced by spaces.

```
ACTTAGCTGACGTTCCGGGTGTTTTTGGCCATCTTCTATAGATTTCCCAGAGACATACTAGGCGTGCTGAAGTTGTGACTCGCGGCCGTATT#CTAACG$T
TTA#TCTCAG#TAGTAG#TGTG#ATTCCCTA$TACTA#TGTCACGCGCGCCCG#TCCT$TTCAGGG#AGTA#CACT#GCGCC#GTAT$GAGCTGAGGTTCG
GGGTGAT#AGGAC#TCCATCGAT#CGTCCTCAGCG#GCTCAAG#CACCCGC#GTTATT$CCGAG#GTGGT#GGTCAT#ATAAATT#CCCCA#TCAA#CCGA
AC$GGTACGG#CCAGCTT#ACACGAGCAGGGC#CTTAAG$CATAGC#GAGG#ACGG#CCACCTTCTATAG#CGAGC#CACCCGC#GTTTTTCCT$CCAAGC
GTC#TGGG#GCGATC#TCTGTAGAGT#CGGAGACAAGCTA#GATGT#TCATTC$CCAAGCTT#TGTACAGT#CTTTGAG#CACACGC$ACAGCG#GGTG#C
GTA#GGATA#GGCAC#GCCTTG$CCTGCTGACGATCGGGGTAGGT#AGGA#TTGAT#GATACAAGCTC#TCTG$TAATATCA#GGCTGG#AGTC#GACTTG
C#GCACTCGT#TCCCTA$GGGATGGG#TAGTCCA#CATGC#CTGCAGTTGTCACTCGTGG#GTGTTCCGTTACG$GGCTGGATTA#TGGTAATC#CGGCCC
T#TTAA$TAATATCAGACGTTCGGGCCGGGTTAGGCCATCTTCTTTAGATTTCTCAGAGACATGCTAGGCGTGCTGAAGTTGGCAATCGCGGACCTGTTCT
CTAACG$
```

Figure 5 The subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer — the 4th-order KATKA kernel — and the copies of $, with maximal omitted substrings replaced by copies of #, except for those adjacent to $.

```
=c<J_cA\2X<G2@'cKNJX5$=c<J_cA\2X\G3K@'cKNJ<5$=c<J_cA\2X\G2@'C6J<
5$=c_cA\2X\G3K@'C6J<5$G__/\<.GC<@CJJ<5$.__CXUGC<@CNJ<.'$G___c=\2
X\.+@CNJ<5$G___92XC.G@'CJJ<5$<<J__.\GG2@QCNJ<5$<<J__.\\\G2@'CNJ<
5$NN__\<J\N2\'+N<5$<.__\<J\N/=N'+J<5$XcN2C<\\\G@2@'+J<5$XcNQC<\\
\GQ2@+C6J<$XcN/A\\\GQ2@'_N\5$XcNJ_\\\GQ2@'+925$
```

Figure 6 Minimizer digests for the toy genomes in Figure 3, separated by $s.

2.3 Minimizer digests

To build a *minimizer digest* [24] for a string $S[1..n]$, we

1. choose parameters k and w and a hash function $h(\cdot)$ function on k-mers,
2. mark each k-mer $S[j..j+k-1]$ in S such that $h(S[j..j+k-1])$ is the leftmost occurrence of the minimum in $h(S[i..i+k-1], \ldots, S[i+w-1..(i+w-1)+k-1])$ for some i with $i \leq j < i+w$,
3. return the sequence of marked k-mers' hashes.

For example, suppose $k=3$, $w=10$ and the hash function $h(\cdot)$ takes a triple over $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ as a 3-digit number x in base 4 and returns $(2544x+3937) \bmod 8863$. The minimizer digests for the toy genomes in Figure 3 (excluding $s) are shown in Figure 6 separated by $s and with the 64 triples over $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ mapped to ASCII values between 37 and 100. Minimizer digests are widely used in bioinformatics to reduce tools' time and space requirements; for example, they are used this way in Kraken 2 [27], mdBG [7] and SPUMONI 2 [2].

We note that although the first minimizer digest `=c<J_cA\2X<G2@'cKNJX5` is 21 characters while the first genome is 100 characters, the digest is over an alphabet of size 64 instead of 4; therefore, the minimizer is 126 bits while the genome is 200 bits. The space of the auxiliary data structures for an augmented FM-index for the minimizer digest still depends on the number 21 of characters in the digest, however.

We say the concatenation of the minimizer digests for the genomes in a collection, separated by $s, is the minimizer digest for the collection. By construction, if α is the minimizer digest for a pattern and there are i copies of $ to the left of the first occurrence of α in the minimizer digest for the collection, then the first occurrence of the pattern cannot be before the $(i+1)$st genome (and symmetrically for the last occurrences) — although the pattern may not occur in that genome and possibly not in the whole collection.

2.4 KATKA kernels of minimizer digests

Of course, we can also build KATKA kernels of minimizer digests. Figure 7 shows the subsequence consisting of the characters in the first or last occurrence of each distinct pair — the 2nd-order KATKA kernel — and the copies of $ in Figure 6, with maximal omitted substrings replaced by copies of #. It consists of 220 6-bit characters (1320 bits) plus the 16 $s; the original minimizer digest consists of 287 6-bit characters (1722 bits) plus the $s, the 4th-order KATKA kernel consists of 798 2-bit characters (1596 bits) plus the $s, and the collection of toy genomes itself consists of 1600 2-bit characters (3200 bits) plus the $s. We note that pairs of minimizers with $k=3$ and $w=10$ can represent substrings as short as 4 characters or as long as 17 characters in the genomes; in our example, on average a pair of minimizers represents about $2 \cdot (1600/287) \approx 11.15$ characters.

KATKA kernels of minimizer digests may inherit the strengths of both: with kernelization we can take advantage of repetition to compress, while using minimizers allows us to keep the parameter k in the kernelization small while still dealing with reasonably long patterns.

```
=c<J_cA\2X<G2@’cKNJX5$X\G3K@’cKNJ<5$c<#’C6J$=c_cA#G3K@$G__/\<.GC
<@CJJ$.__CXUGC<@CN#.’$_c=\2X\.+@C$G_#_92XC.G@#CJJ$<<#_.\GG#@QC$<
<#_.\\#G2#’CN$NN__\#J\N2\’+N<$<.__\<J\N/=N’+J$XcN2C<\#G@2#+J<5$N
QC<\#GQ2@+C6J<$N/A\#’_N\5$XcNJ_\\\GQ2@’+925$
```

Figure 7 The subsequence consisting of the characters in the first or last occurrence of each distinct pair — the 2nd-order KATKA kernel — and the copies of `$` in Figure 6, with maximal omitted substrings replaced by copies of `#`.

3 Approximating MEM tables with FM-indexes of KATKA kernels and minimizer digests

Once we have built a KATKA kernel or minimizer digest for a collection of genomes, or a KATKA kernel of a minimizer digest, we can build an augmented FM-index over it. For example, Figure 8 shows the first and last lines of the augmented FM-indexes for the 4th-order KATKA kernel in Figure 5; the minimizer digest in Figure 6; and the 2nd-order KATKA kernel of the minimizer digest, from Figure 7. In all three cases, we include an implicit end-of-file character less than any other.

Consider the pattern $P =$ `GGATGGGCTAGACGATCTTCTGTG`, which we obtained by choosing the substring `GGGTGGGTTAGACGATCTTCTGTA` of toy genome 9 in Figure 3 (numbering the genomes from 0) and changing two characters. The MEM table of P with respect to all the toy genomes is shown on the left in Figure 9. The MEM table of P with respect to the 4th-order KATKA kernel with `$`s and `#`s shown in Figure 5, is shown in the center of Figure 9. (The MEM table of P with respect to the 5th-order KATKA kernel is the same as its MEM table with respect to the genomes.) The minimizer digest of P with $w = 10$ is `Q.` and the MEM table of that with respect to the minimizer digest of the collection is shown on the right of Figure 9; the MEM table with respect to the 2nd-order KATKA kernel of the minimizer digest is the same as the MEM table with respect to the minimizer digest.

Since P comes from toy genome 9, following Wood, Lu and Langmead's [27] terminology in their presentation of Kraken 2, we classify MEMs' [first, last] ranges as *true positives* if they are exactly [9,9], *false positives* if they exclude 9 but are not empty, *vague positives* if they include 9 and at least one other number, and false negatives if they are empty. The classification of the MEMs' ranges in Figure 9 are shown below:

true positives	false positives	vague positives	false negatives
[9]	[0, 1], [8], [11]	[0, 15], [4, 11]	
	[12, 14], [13], [15]	[6, 15], [8, 15]	

Notice the ranges for MEMs with respect to the toy genomes and the 4th-order KATKA kernel can never be empty (assuming every distinct character in P occurs in the genomes at least once), so those ranges cannot be false negatives. On the other hand, if we generate P by changing characters in a way that disrupts every previous minimizer and creates new minimizers that are not in the minimizer digest of the genomes, then we can get MEMs with respect to the minimizer digest or to the 2nd-order KATKA kernel of the minimizer digest, whose ranges are empty.

Looking at the MEM table of P with respect to the toy genomes, it is intuitive to give more weight to the longer MEM, which occurs only in genome 9. If on this basis we guess correctly that P came from genome 9, then we can consider P a true positive with respect to the toy genomes; unfortunately, the same is not true with respect to the 4th-order KATKA kernel, nor to the minimizer digest with $w = 10$.

i	SA[i]	LCP[i]	BWT[i]	context
0	815	0	$	
1	321	0	T	#ACACGAGCA...
2	354	3	G	#ACGG#CCAC...
3	550	2	T	#AGGA#TTGA...
4	209	5	T	#AGGAC#TCC...
5	167	3	G	#AGTA#CACT...
⋮	⋮	⋮	⋮	⋮

i	SA[i]	LCP[i]	BWT[i]	context
⋮	⋮	⋮	⋮	⋮
810	477	3	C	TTTGAG#CAC...
811	23	4	T	TTTGGCCATC...
812	390	3	T	TTTTCCT$CC...
813	22	4	T	TTTTGGCCAT...
814	389	4	G	TTTTTCCT$C...
815	21	5	G	TTTTTGGCCA...

A	C	T	T	A	G	C	...	C	T	A	A	C	G	$
0	1	2	3	4	5	6		1121	1122	1123	1124	1125	1126	1127

$B =$ 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010
00000000000000000000000000000001000000000000000000000000001000000000000000000000000000001000000000000000 ...
00000000000000000100000000000000000000000000000000000000000000000000000000100000000000000000000000000
00000010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

i	SA[i]	LCP[i]	BWT[i]	context
0	303	0	$	
1	302	0	5	$
2	102	1	5	$.__CXUGC<...
3	210	1	5	$<.__\<J\N...
4	156	2	5	$<<J__.\GG...
5	174	8	5	$<<J__.\\\...
⋮	⋮	⋮	⋮	⋮

i	SA[i]	LCP[i]	BWT[i]	context
⋮	⋮	⋮	⋮	⋮
298	15	4	’	cKNJX5$=c<...
299	268	1	X	cN/A\\\GQ2...
300	230	2	X	cN2C<\\\G@...
301	286	2	X	cNJ_\\\GQ2...
302	249	2	X	cNQC<\\\GQ...
303	67	1	=	c_cA\2X\G3...

=	c	<	J	_	c	A	...	@	’	+	9	2	5	$
0	1	2	3	4	5	6		296	297	298	299	300	301	302

$B =$ 0000000000000000000001000000000000000000 ...
0001000000000000000000000000000000000001

i	SA[i]	LCP[i]	BWT[i]	context
0	236	0	$	
1	38	0	<	#’C6J$=c_c...
2	137	3	2	#’CN$NN__\...
3	211	2	\	#’_N\5$XcN...
4	185	1	2	#+J<5$NQC<...
5	82	1	N	#.’$_c=\2X...
⋮	⋮	⋮	⋮	⋮

i	SA[i]	LCP[i]	BWT[i]	context
⋮	⋮	⋮	⋮	⋮
231	5	2	_	cA\2X<G2@’...
232	29	1	’	cKNJ<5$c<#...
233	15	4	’	cKNJX5$X\G...
234	175	1	X	cN2C<\#G@2...
235	219	2	X	cNJ_\\\GQ2...
236	45	1	=	c_cA#G3K@$...

=	c	<	J	_	c	A	...	@	’	+	9	2	5	$
0	1	2	3	4	5	6		229	230	231	232	233	234	235

$B =$ 0000000000000000000001000000000000010000 ...
0000000000100000000001000000000000000001

Figure 8 The first and last lines of the augmented FM-indexes for the KATKA kernel in Figure 5 **(top)** and the minimizer digest in Figure 6 **(bottom)**.

MEM	first	last
GGATGGGCTAG	13	13
TAGACGATCTTCTGT	9	9
TGTG	0	1

MEM	first	last
GGATGGG	13	13
GGGC	6	15
GGCT	12	14
GCTAG	15	15
TAGA	0	15
AGACG	15	15
GACGATC	11	11
ATCTTCT	0	15
TCTGT	8	8
TGTG	0	1

MEM	first	last
Q	8	15
.	4	11

Figure 9 The MEM tables of P with respect to the toy genomes in Figure 3 (left), the 4th-order KATKA kernel in Figure 5 (center), and the minimizer digests in Figure 6 (right).

4 Experiments

In order to present a concise comparison of results obtained with a full dataset with those obtained with KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests, for this section we focus on true-positive rates rather than whole MEM tables. We classify a read as a true positive if its longest MEM is a true positive (or all its longest MEMs, in the case of a tie).

We wrote the code for our experiments (which computes full MEM tables) in C++ using SDSL [26] and posted it at `https://github.com/draessld/KATKA2`. We ran our experiments on a server at the Department of Computer Science of the Czech Technical University in Prague with 128 AMD EPYC 7742 64-Core CPUs and 504 GiB of RAM, running GNU/Linux Kernel 5.15.0.

We chose 1000 bacterial genera consecutive in the phylogenetic tree for 138.1 release of the SILVA SSU Ref NR99 database [23] of ribosomal RNA (rRNA) gene sequences. We concatenated the gene sequences for the genera, separated by \$s, and built augmented FM-indexes for that 167328343-character concatenation, and KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests for it with various parameter settings:

- for KATKA kernels of the original concatenation, we used $k = 5, 10, 15, 20, \ldots, 45, 50, 100$;
- for minimizer digests, we used 3-mers as minimizers and set $w = 5, 10, 15, 20, \ldots, 45, 50$;
- for KATKA kernels of the minimizer digests, we used $k = 5, 10, 15, 20, \ldots, 45, 50$ and the same w values.

We included the kernel with $k = 100$ of the original concatenation to show that as k increases, the true-positive rate does approach the rate achieved with an index of the original concatenation.

For each genus g, we simulated 500 reads of 200 base pairs each by choosing a random starting location in the reference sequence for g and mutating 1% percent of the bases uniformly across the read to simulate sequencing error. For each read and each index, we found all the read's longest MEMs and checked whether all their [first, last] ranges contained only the ID of the reference sequence for g. Figure 10 shows the index sizes and true-positive rates over all 500 000 simulated reads. Clearly, we can achieve significant compression while only slightly decreasing the true-positive rate, especially with KATKA kernels of minimizer digests: for example, with $k = 30$ and $w = 5$ our index took 56.5 MiB and achieved a true-positive rate of 74.3%, compared to 287.9 MiB and 78.6% with an index for the full dataset, better than the tradeoffs we achieve with kernelization or minimizers alone.

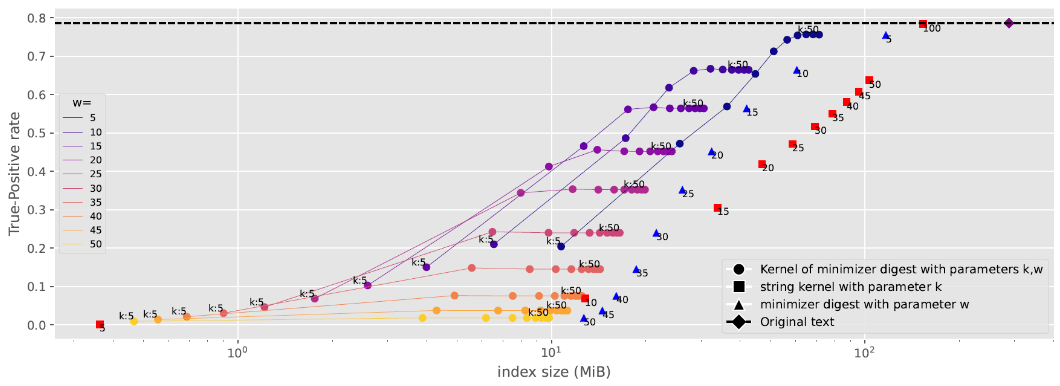


Figure 10 The index size in MiB and the true-positive rate as a percentage, for the original dataset and various KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests.

5 Conclusions and future work

Figure 10 strongly confirms our conjecture from Subsection 2.4 that KATKA kernels of minimizer digests can inherit the strengths of both. In the near future we plan experiment also with varying the width of minimizers (for simplicity, in this paper we always used 3-mers) and to measure also the speedups we can achieve. (Searching over minimizer digests is usually significantly faster than searching over original texts, both because some characters are not represented in the digests and because we use a backward step for each minimizer rather than for each character, incurring fewer cache misses.) Later, we plan to incorporate indexing KATKA kernels of minimizer digests to build MEM tables — with more sophisticated classifications that take advantage of all the information in those tables — into a full pipeline for taxonomic classification of reads.

The confirmation of our conjecture may be useful for other applications as well, when we are dealing with repetitive datasets and want the flexibility of an augmented FM-index (instead of an r-index or a grammar-based index, for example) but kernelization has still had less impact than we might have hoped, because setting the parameter k high enough to allow for the pattern lengths used in practice results in poor compression. For example, an obvious question that arises from our work is whether Valenzuela et al.'s [25] PanVC tool can achieve interesting tradeoffs between compression and accuracy using kernelization of minimizer digests, instead of only kernelization.

References

1 Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *Iscience*, 24(6), 2021.

2 Omar Y Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. SPUMONI 2: improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24(1):122, 2023.

3 Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-Seq quantification. *Nature biotechnology*, 34(5):525–527, 2016.

4 Florian P Breitwieser, Daniel N Baker, and Steven L Salzberg. KrakenUniq: confident and fast metagenomics classification using unique k-mer counts. *Genome biology*, 19(1):1–10, 2018.

5 Manuel Cáceres and Gonzalo Navarro. Faster repetition-aware compressed suffix trees based on block trees. *Information and Computation*, 285:104749, 2022.

6 Marie Cheng, Omar Ahmed, Anna Liebhoff, and Ben Langmead. Factors affecting k-mer specificity and alternative approaches for metagenomic classification. In preparation.

7 Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell systems*, 12(10):958–968, 2021.

8 Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130137, 2014.

9 Héctor Ferrada, Dominik Kempa, and Simon J Puglisi. Hybrid indexing revisited. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8. SIAM, 2018.

10 Travis Gagie, Sana Kashgouli, and Ben Langmead. KATKA: A KRAKEN-like tool with k given at query time. In *International Symposium on String Processing and Information Retrieval*, pages 191–197. Springer, 2022.

11 Travis Gagie and Simon J Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

12 Younan Gao. Computing matching statistics on repetitive texts. In *2022 Data Compression Conference (DCC)*, pages 73–82. IEEE, 2022.

13 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014. Proceedings 13*, pages 326–337. Springer, 2014.

14 Daehwan Kim, Li Song, Florian P Breitwieser, and Steven L Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome research*, 26(12):1721–1729, 2016.

15 Sam Kovaka, Yunfan Fan, Bohan Ni, Winston Timp, and Michael C Schatz. Targeted nanopore sequencing by real-time mapping of raw electrical signal with UNCALLED. *Nature biotechnology*, 39(4):431–441, 2021.

16 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

17 Peter Menzel, Kim Lee Ng, and Anders Krogh. Fast and sensitive taxonomic classification for metagenomics with Kaiju. *Nature communications*, 7(1):11257, 2016.

18 Daniel J Nasko, Sergey Koren, Adam M Phillippy, and Todd J Treangen. RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. *Genome biology*, 19(1):1–10, 2018.

19 Gonzalo Navarro. Computing MEMs on repetitive text collections. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

20 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String Processing and Information Retrieval: 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings 17*, pages 347–358. Springer, 2010.

21 Vitor C Piro, Martin S Lindner, and Bernhard Y Renard. DUDes: a top-down taxonomic profiler for metagenomics. *Bioinformatics*, 32(15):2272–2280, 2016.

22 Petr Procházka and Jan Holub. Compressing similar biological sequences using FM-index. In *2014 Data Compression Conference*, pages 312–321. IEEE, 2014.

23 Christian Quast, Elmar Pruesse, Pelin Yilmaz, Jan Gerken, Timmy Schweer, Pablo Yarza, Jörg Peplies, and Frank Oliver Glöckner. The silva ribosomal rna gene database project: improved data processing and web-based tools. *Nucleic acids research*, 41(D1):D590–D596, 2012.

24 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.

25 Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC genomics*, 19(2):123–130, 2018.

26 vgteam. sdsl-lite. `https://github.com/vgteam/sdsl-lite`, 2022.

27 Derrick E Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with Kraken 2. *Genome biology*, 20:1–13, 2019.

28 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1–12, 2014.