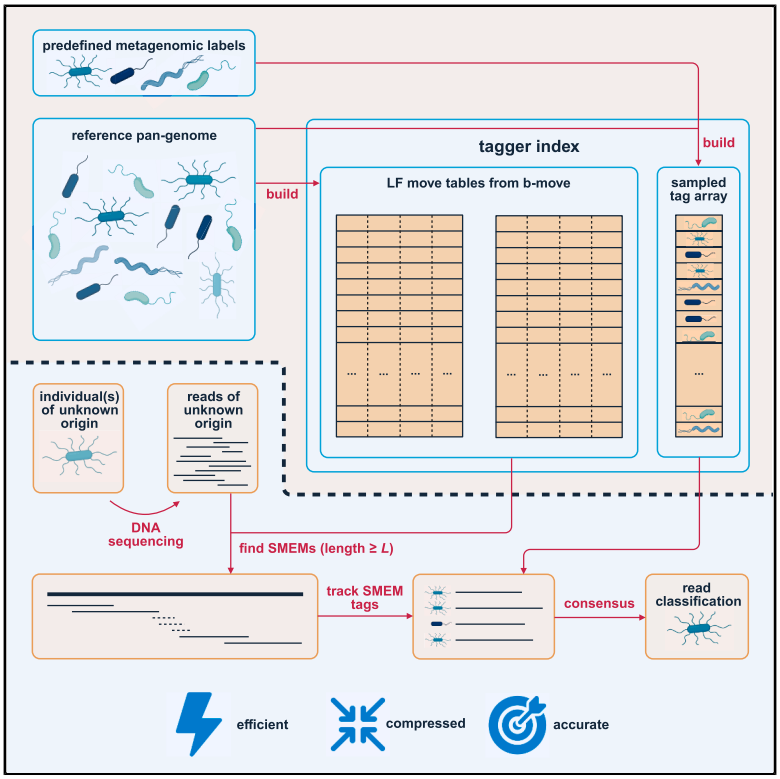
iScience

Article

Run-length compressed metagenomic read classification with SMEM-finding and tagging

Graphical abstract



Highlights

- First run-length compressed classifier based on full SMEMs instead of semi-SMEMs
- Sampled tag arrays enable fast and memory-efficient multi-class classification
- High accuracy on both long and short reads across diverse metagenomic datasets
- More balanced in accuracy, runtime, and memory than other run-length compression tools

Authors

Lore Depuydt, Omar Y. Ahmed, Jan Fostier, Ben Langmead, Travis Gagie

Correspondence

lore.depuydt@gmail.com

In brief

Microbial genomics; Biological classification; Biocomputational method; Classification of bioinformatical subject

Depuydt et al., 2025, iScience *28*, 114029
December 19, 2025
https://doi.org/10.1016/j.isci.2025.114029

CellPress

iScience

CellPress
OPEN ACCESS

Article

Run-length compressed metagenomic read classification with SMEM-finding and tagging

Lore Depuydt,[1,4,*] Omar Y. Ahmed,[2] Jan Fostier,[1] Ben Langmead,[2] and Travis Gagie[3]
[1]Department of Information Technology - IDLab, Ghent University - imec, 9052 Gent, Belgium
[2]Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA
[3]Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 4R2, Canada
[4]Lead contact
*Correspondence: lore.depuydt@gmail.com
https://doi.org/10.1016/j.isci.2025.114029

SUMMARY

Metagenomic read classification is a fundamental task in computational biology but remains challenging due to the scale and diversity of sequencing data. We present a run-length compressed BWT-based index using the move structure for efficient multi-class classification. Our method finds all super-maximal exact matches (SMEMs) of length $\geq L$ between a read and a reference and associates each SMEM with one class identifier using a sampled tag array. A consensus algorithm then compacts these SMEMs and their class identifiers into a single classification. We are the first to perform run-length compressed read classification using full rather than semi-SMEMs. We evaluated on long and short reads across two datasets: a large bacterial pan-genome with few classes and a smaller 16S rRNA gene database spanning thousands of genera. Our method outperforms SPUMONI 2 in accuracy and runtime while maintaining run-length compressed memory complexity and surpasses Cliffy in memory efficiency with comparable accuracy.

INTRODUCTION

Metagenomic read classification, where reads of unknown origin are matched against a reference set of candidate genomes or genes, has been a long-standing challenge in bioinformatics due to computational complexity, large and incomplete reference datasets, and the inherent noisiness of sequencing reads. Reads can be classified at different taxonomic levels—such as strain, species, or genus—depending on the question at hand. Metagenomic analyses are revolutionizing diverse fields, including infectious disease diagnostics,[1] cancer prediction,[2] antibiotic resistance surveillance,[3] ecology research,[4] and sustainable agriculture.[5]

Many established metagenomic classification tools exist, such as Kraken 2,[6] Centrifuge,[7] and MetaPhlAn,[8,9] each employing distinct approaches to classification. Kraken 2 uses a probabilistic, compact hash table that maps minimizers onto the lowest common ancestor (LCA) taxa.[6] Centrifuge employs an FM-index,[10] based on the Burrows-Wheeler transform (BWT),[11] to find exact matches between reads and candidate references.[7] MetaPhlAn(4) relies on a preprocessed subset of clade-specific marker genes for classification.[8,9] However, as reference databases continue to grow, these tools face scalability challenges. Lossless tools such as Centrifuge experience index growth proportional to the sequence content in the reference, which becomes impractical even for high-RAM workstations. Lossy tools such as Kraken 2 and MetaPhlAn lose specificity as their database size increases.[12] More specifically, for *k-mer*-based tools such as Kraken 2, there is no single value of k that consistently yields optimal results across all scenarios.[12,13]

Orthogonally, recent advancements have introduced a new wave of bioinformatics tools leveraging the compressibility of the BWT to create compact and efficient indexes, particularly for large, repetitive datasets. This approach was first applied to the run-length FM-index (RLFM-index)[14] and the r-index,[15,16] both achieving $O(r)$ space complexity, where r is the number of character runs in the BWT. The introduction of the move structure[17] further accelerated this functionality by supporting $O(1)$-time core LF operations while maintaining $O(r)$ memory complexity. Building on these foundations, alignment tools such as MONI,[18] PHONI,[19] SPUMONI,[20] Movi,[21] and b-move[22,23] have demonstrated the practical utility of run-length compressed indexes. These tools focus on tasks such as computing matching statistics or pseudomatching lengths, finding and extending maximal exact matches, or lossless approximate pattern matching.

In metagenomic classification, tools such as SPUMONI 2,[24] Centrifuger,[25] and Cliffy[26] have also applied run-length compression to create memory-efficient solutions. SPUMONI 2 performs multi-class metagenomic classification in $O(r)$ space using a sampled document array and pseudomatching lengths, based on MONI's matching statistics.[18,24] Optional lossy minimizer digestion can further accelerate and compress the process with minimal accuracy penalties. Centrifuger supports taxonomic classification, i.e., mapping reads to the LCA of their matches in the taxonomic tree. It uses a lossless run-block compressed BWT of

the reference to find semi-maximal matches between reads and references,[25] though its worst-case space complexity remains $O(n)$. Cliffy, though mainly intended for taxonomic classification, also supports multi-class metagenomic classification (referred to as document listing) using a run-length compressed BWT with document listing profiles.[27] It achieves an $O(rd)$ space complexity, where d is the number of classes. Optionally, Cliffy's lossy cliff compression significantly reduces index size while still supporting *approximate* document listing and accurate taxonomic classification.[26]

Though these tools demonstrate the potential of run-length compressed indexes in metagenomic classification, our experiments show that there is still a need for further improvements in balancing space efficiency, scalability, performance, and classification accuracy. One limitation is that these tools build upon the r-index's pattern matching principles, which can suffer significant computational overhead due to the multitude of cache misses introduced by complex memory access patterns.[22] Moreover, they all perform backward search to find semi-maximal matches but lack support for *full* maximal exact match-based classification. By considering only semi-maximal matches, specificity is reduced, which is particularly critical for distinguishing closely related species or even strains lower in the phylogenetic tree. In contrast, full maximal exact matches ensure that all the reported matches are truly non-extendable, reducing ambiguity in classification.

Contributions

We propose an index that efficiently supports multi-class metagenomic classification in run-length compressed space. Classification begins by identifying all *full* super-maximal exact matches (SMEMs) of a read relative to the reference dataset, where an SMEM is defined as an exact match that cannot be extended in either direction of the read while still occurring in the reference dataset.[28] The ability to compute all SMEMs encompasses all other exact matching queries, including various forms of half-maximal match queries and k-mer searches, as these can all be derived from the full set of SMEMs. To improve specificity, only SMEMs of length at least L are considered, found efficiently using a method similar to that described by Gagie et al.[29] (see also Li's ropebwt3[30]), illustrated in Figure 1.

During SMEM identification, we use a sampled tag array (e.g., Table 1) of class identifiers[31] to track the identifier of *one* metagenomic class in which the SMEM occurs. To maintain run-length compressed space complexity, the tag array is sampled at positions marking the end of a BWT run. After identifying SMEMs and their associated class identifiers, a consensus algorithm aggregates the results into a single classification per read. In addition to these theoretical advancements in memory usage and accuracy, we leverage a move structure-based index (e.g., Tables 1 and 2) for high performance, irrespective of theoretical time complexities.

We evaluate our approach in terms of accuracy, runtime, and memory usage by comparing it to SPUMONI 2 and Cliffy and observe that our approach achieves the best balance across all metrics. Its versatility is demonstrated by classifying both long and short reads on two conceptually different datasets. Kraken 2 is also included in the evaluation to provide a baseline for state-of-the-art metagenomic read classification.

RESULTS

Setup and hardware

We compare the accuracy and performance of our metagenomic read classification implementation with SPUMONI 2[24] and Cliffy,[26] the only run-length compressed tools, to our knowledge, supporting multi-class metagenomic read classification without relying on taxonomy. SPUMONI 2 can build and query indexes with and without minimizer digestion (`SPUMONI v2.0.7: spumoni build/run --PML --doc-array --no-digest/--minimizer-alphabet`). Minimizer digestion reduces runtime and memory usage but introduces a slight accuracy penalty due to the use of a lossy index. Both configurations are evaluated. SPUMONI 2 produces pseudomatching lengths, each with *one* corresponding class identifier, for each position in the read P. From these $|P|$ identifiers, we determine the consensus class using a majority vote, as described in the SPUMONI 2 paper.[24] Post-processing majority vote runtimes are excluded from benchmarks.

Cliffy, though mainly intended for taxonomic classification, also supports multi-class metagenomic classification (`cliffy v2.0.0: cliffy build --revcomp --two-pass [--minimizers] [--taxcomp --num-col 7]; cliffy run --ftab [--minimizers] [--taxcomp --num-col 7]`). Cliffy offers configurations with or without minimizer digestion, with similar trade-offs to SPUMONI 2. Orthogonally, Cliffy supports uncompressed and cliff compressed document array profiles. Uncompressed profiles enable *exact* document listing but are practical only for datasets with few metagenomic classes. To support many metagenomic classes, cliff compression is required to maintain a manageable index size, resulting in *approximate* document listing. Depending on the dataset, we will use the Cliffy configurations that result in a reasonably sized index. For each read, Cliffy produces variable-length exact matches, each with a set of candidate classes. We assign a single metagenomic class to each read using a consensus algorithm that weighs candidate classes by match length and the number of classes per match, following the method by Ahmed et al.[26] This consensus algorithm is excluded from runtime benchmarks.

Furthermore, we compare our approach to Kraken 2's accuracy and performance, establishing a baseline for current state-of-the-art read classification.[6] Our goal is not to outperform Kraken 2 in all aspects but to demonstrate the potential of SMEM-finding and tagging on a run-length compressed BWT-based index for bioinformatics applications such as metagenomic classification. Maintaining perspective on leading methods remains important to evaluate where our approach can offer improvements. Kraken 2 uses a compact hash table to store minimizer-taxonomy ID pairs, reducing memory by subsampling k-mers to their smallest canonical ℓ-mer minimizers. We used default parameters: $k = 35$ and $\ell = 31$ (`Kraken 2 v2.1.5: default parameters`). Kraken 2 classifies sequences by querying minimizers against the hash table, assigning taxonomy based on the lowest common ancestor in a probabilistic, space-efficient manner. To enable comparison

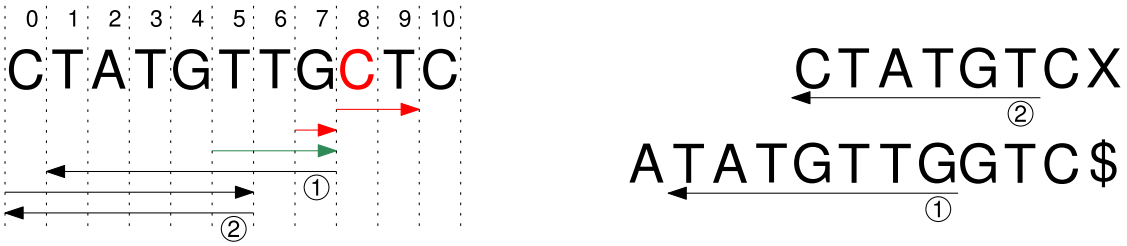


Figure 1. Visualization of SMEM search steps in an example read

Left: visualization of the search steps in Algorithm 2 to execute `findSMEMs(P,M,M^rev,3)`, where P = "CTATGTTGCTC," and M and M^{rev} correspond to the example move tables in Tables 1 and 2. $P[8] = C$ represents a sequencing error. The red arrows (top two) represent failed jump start searches, and the green arrow (third from the top) represents a successful jump start search. Black arrows (bottom three) represent searches that have characterize valid SMEMs of length at least $L = 3$. Right: visualization of the detected SMEMs in the search text T.

with multi-class (non-taxonomic) classifiers, we built Kraken 2 indexes as a flat hierarchy, allowing reads to be classified only to the root or a leaf, or left unclassified.

All benchmarks used a single core of two 96-core AMD EPYC 9654 CPUs (2.40 GHz) and 370 GiB of RAM. Runtimes represent the median of 20 runs and exclude index loading times.

Case study with 8,165 bacterial genomes

We performed a case study similar to the SPUMONI 2 multi-class classification experiment.[24] A pan-genome was created by combining all RefSeq strains from eight bacterial species: *Escherichia coli*, *Salmonella enterica*, *Listeria monocytogenes*, *Pseudomonas aeruginosa*, *Bacillus subtilis*, *Limosilactobacillus fermentum*, *Enterococcus faecalis*, and *Staphylococcus aureus*. This pan-genome contains 8,165 sequences (details available at https://github.com/biointec/tagger/tree/data/BacterialGenomes) across 8 metagenomic classes, totaling 37.4 Gbp. To assess read classification accuracy, we simulated 50,000 long reads (`PBSIM v2.0.1: pbsim --hmm_model R94.model --accuracy-mean 0.95`)[32] (average length 5,236 bp) and 500,000 short reads (`ART_Illumina v2.3.7: art_illumina --seqSys MS --len 250`)[33] (length 250 bp) per species.

Figure 2 compares the accuracy and runtime performance of our approach against SPUMONI 2, Cliffy, and Kraken 2 for this dataset (an extended version of the bottom left plot is shown in Figure S1). For Cliffy, only the index with minimizer digests is included, as its counterpart without minimizers exceeds our workstation's RAM capacity. Both uncompressed and cliff compressed document array profiles can be analyzed due to the small number of metagenomic classes. Our approach (for optimal L) outperforms SPUMONI 2, Cliffy, and Kraken 2 in accuracy across all configurations, as shown in the left panels. While the improvement over SPUMONI 2 and Kraken 2 is modest, especially for short reads, Cliffy demonstrates noticeably lower accuracy overall. Specifically, cliff compressed document array profiles show a significant reduction in accuracy for this dataset, particularly for long reads. This indicates that cliff compression is not suited for datasets with few metagenomic classes.

In terms of runtime (right panels), our approach outperforms SPUMONI 2 when L is sufficiently large ($L \geq 21$ for long reads and SPUMONI 2 using minimizer digests, or even smaller in other cases). Conveniently, this value of L aligns with our best classification accuracy (also for short reads, accuracy is 99.85% at L = 21). We are unable to beat Cliffy's runtime; however, given Cliffy's significantly lower accuracy, runtime comparisons are less meaningful. Kraken 2's hash-based approach remains an order of magnitude faster than ours for a well-chosen value of L.

Per-class analysis

To assess potential bias, such as toward more abundant classes, we analyze the accuracy, sensitivity, specificity, and precision per class for the long read experiment with optimal parameter $L = 21$ (see Figure S2). Our results show that the approach performs consistently across all 8 species in the classification task, regardless of abundance. The most noticeable deviations are lower precision for *E. coli* and lower sensitivity for *S. enterica*, which arise from 161 *S. enterica* reads being misclassified as *E. coli*. Since *S. enterica* is closely related to *E. coli* but possesses a significant number of additional virulence genes,[34] these misclassifications are not unexpected.

Super-maximal exact match length distribution

To demonstrate the benefit of restricting the SMEM search to a minimum length of L, we analyze the distribution of the lengths of all SMEMs found during the long read experiment for $L = 1$ (see Figure S3). We also visualize the fraction of these SMEMs that have a correct tag toehold with respect to the read they originate from. We observe a significant peak in SMEMs shorter than 20 bases, with the highest frequency at length 14. Moreover, most of these short SMEMs correspond to incorrect tag toeholds. This confirms that, beyond reducing runtime, setting a minimum SMEM length of $L \geq 20$ improves SMEM precision.

Case study with 402,378 16S rRNA genes

To demonstrate the versatility of our approach, we also evaluated its performance on the SILVA SSU NR99 (version 138.1) database, which contains 510,508 16S rRNA sequences, similar to Cliffy's analysis.[26] From this collection, we selected the 402,378 16S rRNA gene sequences (Details available at https://github.com/biointec/tagger/tree/data/SILVA) annotated to the genus level, totaling to 586 Mbp of data spanning 9,118 genera, which serve as the metagenomic classes. Compared to the bacterial pan-genome dataset, this dataset is smaller in size but contains significantly more metagenomic classes. We simulated 5,726,966 long reads (average length 1,415 bp) and 9,029,669 short reads (length

Table 1. Move table for the forward search text

j	c	p	π	ξ
0	C	0	4	2
1	X	1	19	13
2	T	2	11	7
3	$	5	0	0
4	T	6	14	9
5	G	8	7	4
6	T	9	16	11
7	C	11	5	3
8	A	12	1	1
9	G	13	8	5
10	T	15	18	12
11	A	16	2	2
12	G	18	10	6
13	C	19	6	4

Move table M for the search text T = "CTATGTCXATATGTTGGTC$". The corresponding FM-index is provided in Table S1.

Table 2. Move table for the reverse search text

j	c	p	π	ξ
0	C	0	4	1
1	T	1	11	5
2	$	5	0	0
3	X	6	19	11
4	T	7	15	7
5	G	10	7	4
6	A	13	1	1
7	C	15	5	2
8	T	16	18	10
9	C	17	6	3
10	G	18	10	5
11	A	19	3	1

Move table M^{rev} for the reverse search text T^{rev} = "$CTGGTTGTA-TAXCTGTATC". The corresponding FM-index is provided in Table S2.

250 bp), aiming for equal coverage across genera. Figure 3 compares our approach to SPUMONI 2, Cliffy, and Kraken 2. (An extended version of the bottom left plot is provided in Figure S4). For the SILVA dataset, only Cliffy indexes with cliff compressed document array profiles are considered, as their uncompressed counterparts exceed 1 TiB of memory.[26]

For classification accuracy, our approach (for optimal L) consistently outperforms SPUMONI 2 and Kraken 2 for both read types. The improvement over Kraken 2 specifically demonstrates the advantage of using full SMEMs over fixed k-mers; also, at $L = 35$ (in correspondence with Kraken 2's $k = 35$), the accuracy improvement remains significant. However, Cliffy without minimizers achieves the highest accuracy across all tools. For long reads, Cliffy with minimizers also outperforms our method. This suggests that for datasets with many similar metagenomic classes, Cliffy's ability to report multiple classes per match provides a significant advantage.

In terms of runtime, our method again outperforms SPUMONI 2 across all configurations ($L \geq 25$ for long reads and SPUMONI 2 using minimizer digests, or even smaller in other cases). For this dataset, we also surpass Cliffy in runtime (for optimal L), except when matching long reads with $L \leq 47$, where Cliffy with minimizer digests achieves both higher accuracy and faster performance. Kraken 2 is almost an order of magnitude faster than our approach for the optimal value of L.

Memory versus runtime analysis

Finally, we compare the tools in terms of peak memory usage and runtime performance. Figure 4 shows memory versus runtime for all long read experiments (see previous results sections), and Figure S5 covers short reads. For our approach, we fix L at 25, balancing accuracy and runtime as demonstrated above.

For the bacterial genomes dataset, where Cliffy shows significantly lower accuracy (see previous Results sections), Cliffy (uncompressed and cliff compressed) uses 2.4× and 11.5× more memory than our method, respectively. In combination with Cliffy's poor accuracy on this dataset, particularly with cliff compression, these results confirm that Cliffy is less suitable for larger datasets with few metagenomic classes, especially when cliff compression is involved. Compared to SPUMONI 2 (without and with minimizer digests), our method uses 4.1× and 8.7× more memory, respectively, but is faster and more accurate. Our index, under 8 GiB, still fits in standard laptop RAM.

For the SILVA rRNA dataset, the memory difference between Cliffy and the other tools is even more pronounced. Cliffy requires 41.8× and 21.6× more memory than our method (without and with minimizer digests, respectively). Cliffy's higher accuracy (see previous results sections) thus comes at the cost of large document array profiles. For instance, Cliffy (with minimizer digests) is more accurate and faster in one case (Figure 3, top right) but requires nearly 20 GiB of RAM, while our approach uses less than 1 GiB. This could affect the feasibility of running Cliffy on a portable laptop. Compared to SPUMONI 2 (without and with minimizer digests), our method requires 2.2× and 3.7× more memory, respectively, but is again (slightly) faster and more accurate. Both our index and SPUMONI 2's indexes fit within 1 GiB, however.

For both datasets, Kraken 2's hash-based approach offers a favorable runtime-memory trade-off. As noted in the previous sections, it is consistently the fastest method, typically about an order of magnitude faster than BWT-based approaches. In terms of memory usage, Kraken 2 performs best on the bacterial genome dataset, which has the largest sequence content. For the SILVA dataset, however, our approach and both SPUMONI 2 configurations require less memory than Kraken 2. Notably, Kraken 2's SILVA index is twice the size of its bacterial genome index, because the database size scales with the number of taxonomic categories in Kraken 2, unlike in our method or SPUMONI 2.

Classification analysis in a PacBio mock community

To further evaluate classification performance in a realistic setting, we analyze a mock community of 9 strains from 7 bacterial species: 2 strains of *Escherichia coli*, 2 of *Staphylococcus aureus*,

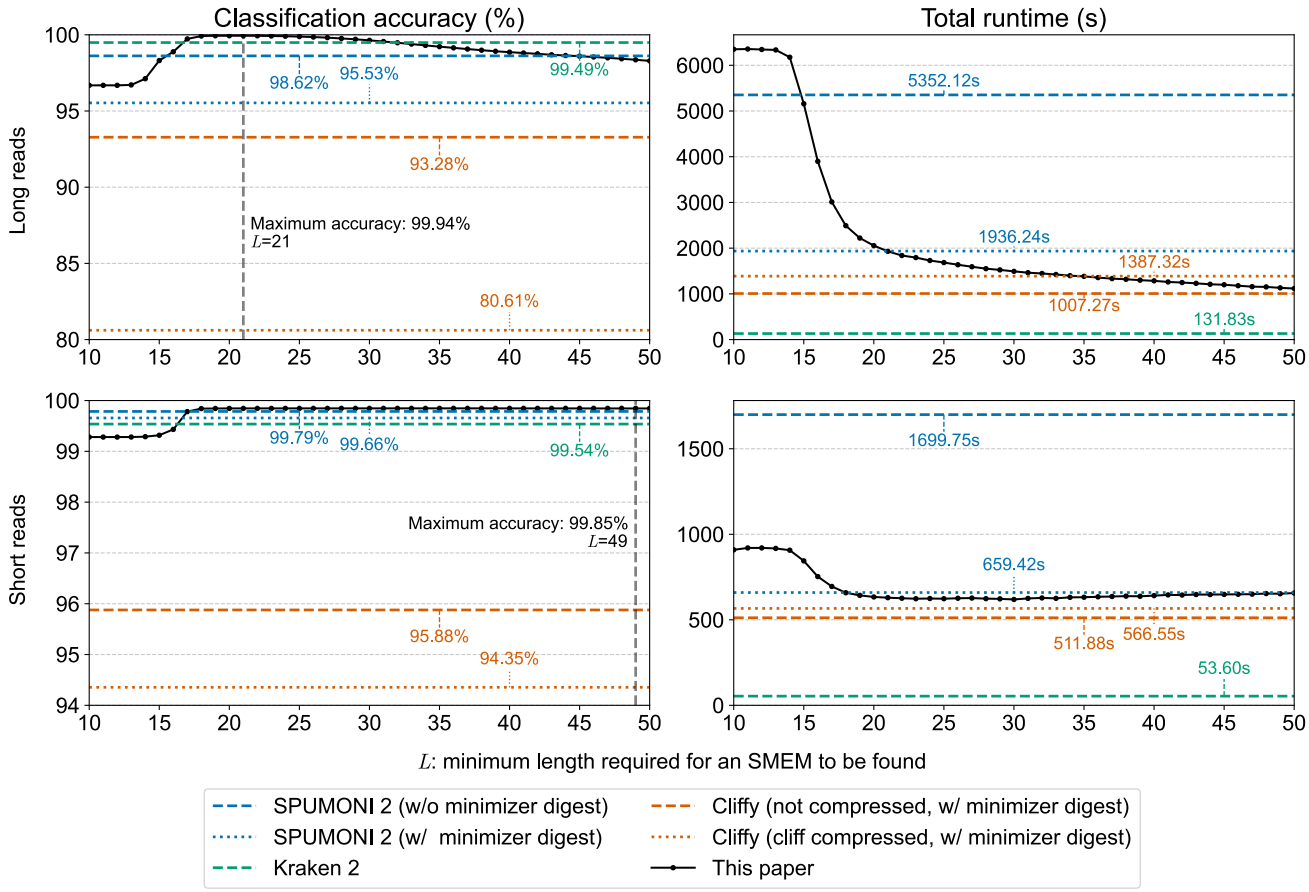


Figure 2. Metagenomic classification of long and short reads across 8,165 bacterial genomes

Analysis of metagenomic classification accuracy (left) and runtime performance (right) for matching 400,000 long reads (top) and 4,000,000 short reads (bottom) to 8,165 bacterial genomes across 8 metagenomic classes. We compare our approach for varying values of L to SPUMONI 2 (without/with minimizer digests), Cliffy with minimizer digests (with uncompressed/cliff compressed document array profiles), and Kraken 2. Kraken 2 classified 31 long and 17,361 short reads to the root, which is considered incorrect. Runtimes represent the median of 20 runs and exclude index loading times.

and 1 each of *Bacillus cereus*, *Klebsiella pneumoniae*, *Listeria monocytogenes*, *Neisseria meningitidis*, and *Shigella sonnei*. We refer to this dataset as Mock9, following Vicedomini et al.[35] It contains 2 586 933 PacBio reads from a 10-plex PacBio Sequel dataset (https://github.com/PacificBiosciences/DevNet/wiki/Microbial-Multiplexing:-PacBio-Sequel-System,-Chemistry—v3.0,-Analysis—SMRT-Link-v6.0.0), which we demultiplexed per strain based on PacBio barcodes with lima (`lima v2.13.0: lima --same --split-named`).[36]

Of the 7 species in Mock9, 3 occur in our bacterial genomes dataset, while 4 do not. Aligning Mock9 reads to the bacterial genomes dataset therefore mimics a realistic scenario in which a read set contains species both present in and absent from the reference.

Figure 5 summarizes the classification distributions for the Mock9 species obtained with our approach for minimum SMEM lengths of $L = 25$ and $L = 35$. Figure S7 shows the corresponding results for SPUMONI 2 and Cliffy, using all configurations the dataset allows, and Kraken 2.

For the three species present in the reference dataset (top three rows in Figures 5 and S7), our approach classified nearly all reads correctly. Kraken 2 shows the same behavior, whereas SPUMONI 2 and Cliffy often misclassify a substantial fraction of reads to the most abundant class in the reference, *E. coli*.

For the four species absent from the reference dataset (bottom four rows in Figures 5 and S7), increasing L from 25 to 35 improves our method's ability to report them as unclassified. This parameter can be tuned by the user without rebuilding the index. Most misclassified reads in this category are assigned to the more abundant classes. When performing the same analysis for the other tools in Figure S7, Kraken 2 (with $k = 35$) also reports a high fraction of reads as unclassified. In contrast, SPUMONI 2 and Cliffy do not show this behavior, since they consider any match length, however short, when classifying reads. As a result, they assign nearly all reads to the most abundant class in the dataset. As such, our approach combines the strengths of both Kraken 2's *k*-mer method and BWT-based tools: it ignores short, spurious matches that lead to incorrect classification, such as Kraken 2, while also exploiting the additional evidence provided by longer maximal exact matches, which are not captured by *k-mer*-based methods.

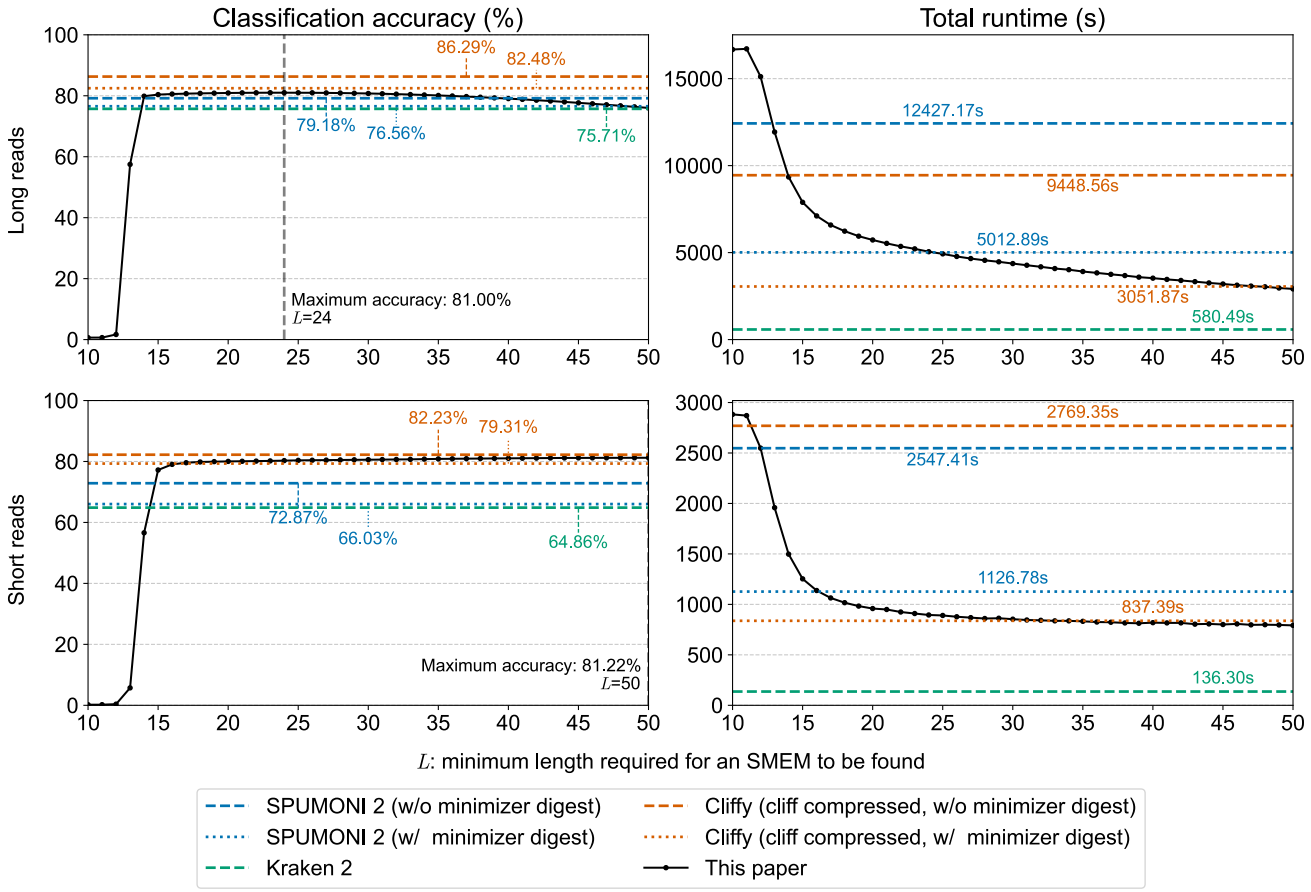


Figure 3. Metagenomic classification of long and short reads from the SILVA dataset

Analysis of metagenomic classification accuracy (left) and runtime performance (right) for matching 5,726,966 long reads (top) and 9,029,669 short reads (bottom) to 402,378 16S rRNA gene sequences spanning 9,118 metagenomic classes. We compare our approach for varying values of L to SPUMONI 2 (without/with minimizer digests), Cliffy with cliff compressed document array profiles (without/with minimizer digests), and Kraken 2. Kraken 2 classified 264,243 long and 2,123,693 short reads to the root, here considered incorrect. Runtimes represent the median of 20 runs and exclude index loading times.

All tools and configurations misclassify nearly all *S. sonnei* reads as *E. coli*, which is unsurprising given their close relatedness within the family *Enterobacteriaceae*.[37] Determining whether any method can reliably separate them when both are in the reference would require further study.

For completeness, Figure S6 reports memory and runtime requirements for all tools and configurations tested here, showing trends similar to those in the previous section.

DISCUSSION

Index construction

Since the index is constructed only once, less time was spent optimizing this part of the implementation. We support two types of index construction: one based on the full suffix array and another using prefix-free parsing.[38] The suffix array approach is faster but memory-intensive, suitable for smaller datasets such as the SILVA dataset (under 1 h runtime, 6.2 GiB peak RAM). Prefix-free parsing is slower but uses less memory, making it ideal for larger pan-genomes such as the bacterial genomes dataset (approximately 13 h runtime, 106 GiB peak RAM, feasible on a regular workstation). We aim to improve our construction implementation based on recent advancements such as grlBWT[39] and ropebwt3.[30]

Alternate or additional implementations

In addition to the algorithms presented above, we explored several alternate implementations, details of which are beyond the scope of this article, that were outperformed by our current approach. One alternative sampled the tag array at the ends of tag runs instead of BWT runs, using a sparse bit vector to mark these sampled positions. This bit vector could identify SMEMs where all occurrences correspond to the same metagenomic class. Another modification combined this bit vector with the tag array sampled at BWT run ends to retain information on SMEMs with a unique class identifier. However, including information on unique SMEMs did not improve accuracy and was thus omitted. We also considered full read alignment for short reads[40] instead of finding SMEMs, but this did not yield better performance or accuracy.

Additionally, we support subsampling the tag array to reduce its memory usage, at the cost of a slight runtime

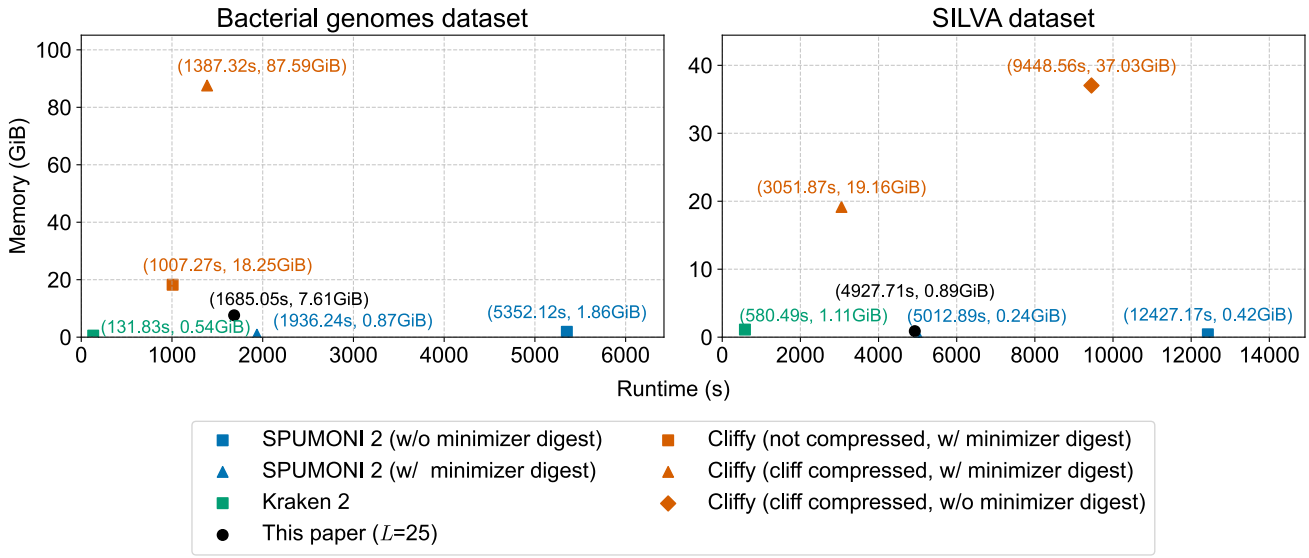


Figure 4. Memory versus runtime comparison on long read datasets

Memory versus runtime comparison of our approach (L = 25) with SPUMONI 2 and Cliffy, using all configurations the dataset allows, and Kraken 2. Shown are results for matching 400,000 long reads to an index of 8,165 bacterial genomes across 8 metagenomic classes (left) and for matching 5,726,966 long reads to an index of 402,378 16S rRNA gene sequences spanning 9,118 metagenomic classes (right). Runtimes represent the median of 20 runs and exclude index loading times.

performance penalty, similar to subsampling the suffix array samples in the sr-index.[41,42] However, the relative memory reduction is modest, as the majority of the index consists of two move tables. Finally, we also support parallelization.

Conclusion

We introduced a run-length compressed approach to metagenomic read classification based on the move structure, SMEM-finding, and sampled tag arrays. Our method demonstrates versatility, achieving high accuracy across various read types (long and short) and datasets, including a large pan-genome with few metagenomic classes and a smaller rRNA database with thousands of classes. We consistently outperform SPUMONI 2, being both faster and more accurate, while maintaining run-length compressed memory complexity. While Cliffy achieves slightly higher accuracy for the complex dataset with thousands of metagenomic classes, this comes at the cost of a significant memory

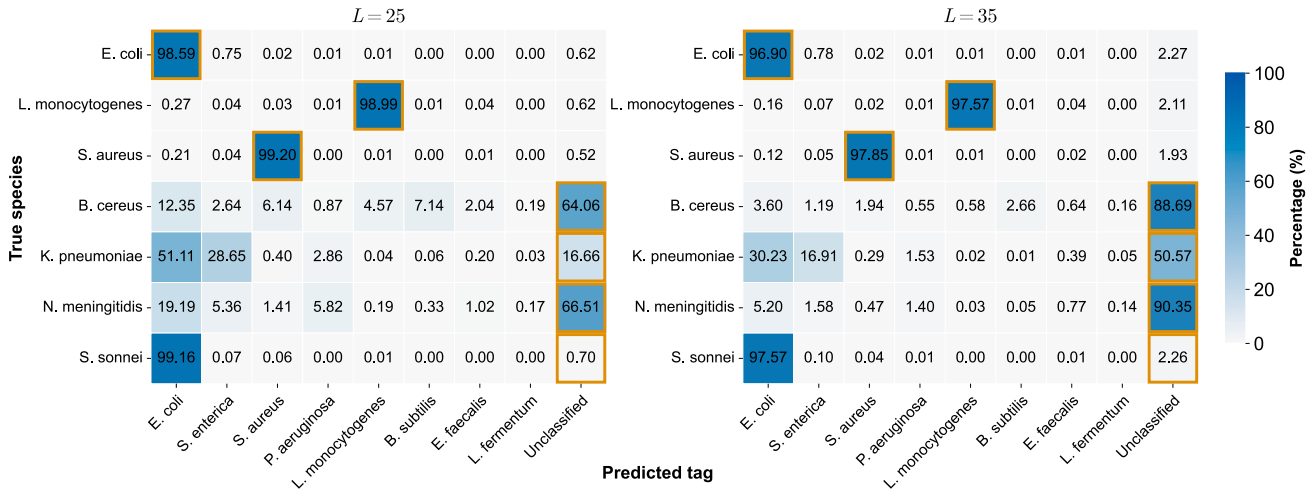


Figure 5. Mock9 classification summary (% per tag per species)

Heatmaps show the classification distribution (% of reads) per predicted tag (columns) for each true species (rows) in the Mock9 dataset. Tags on the *x*-axis (columns) are ordered by decreasing abundance in the reference dataset. Two different minimum SMEM lengths are compared: L = 25 (left) and L = 35 (right). Each cell indicates the percentage of reads from a given species assigned to a specific tag. Orange-outlined boxes highlight the correct tag for each species.

Algorithm 1. Given a pattern *P*, a start (or end) position of interest, and the move table M^{rev} (or *M*), this algorithm finds the corresponding SMEM's end (or start) position using forward (or backward) pattern matching.

```
1  def findSMEMendPos(P, M^rev, startPos):
2      interval ← ([0, n), 0, r^rev)
3      nextPosToMatch ← startPos
4      while interval is not empty and
5      nextPosToMatch < |P| do
6          c ← P[nextPosToMatch]
7          interval ← M^rev.addChar(interval, c)
8          nextPosToMatch ← nextPosToMatch + 1
9      return nextPosToMatch
```

```
def findSMEMstartPos(P, M, endPos):
    interval ← ([0, n), 0, r)
    nextPosToMatch ← endPos − 1
    while interval is not empty and
    nextPosToMatch ≥ 0 do
        c ← P[nextPosToMatch]
        interval ← M.addChar(interval, c)
        nextPosToMatch ←
        nextPosToMatch − 1
    return nextPosToMatch + 1
```

increase (20–40 times more). Kraken 2 remains about an order of magnitude faster than any BWT-based approach, but BWT-based methods' consideration of longer matches can lead to improved accuracy scores. Thus, our approach offers a balanced solution, demonstrating strong performance in accuracy, runtime performance, and memory efficiency, making it suitable for a wide range of metagenomic classification tasks. Our open-source C++11 implementation is available at https://github.com/biointec/tagger under the AGPL-3.0 license.

Limitations of the study

Our current index has not yet been tested on large, taxonomically comprehensive datasets such as GTDB[43,44] and AllTheBacteria.[45] As a result, its scalability and efficiency for such data remain to be demonstrated. Furthermore, the index currently lacks the integration of phylogenetic tree information, which limits its ability to perform advanced, dynamic taxonomic classification. Addressing these limitations will be a focus of future work.

Algorithm 2. This algorithm finds all SMEMs of length at least *L* between pattern *P* and search text *T*, using its move tables *M* and M^{rev}.

```
10 def findSMEMs(P, M, M^rev, L):
11     foundSMEMs ← {}
12     currentStartPos, currentEndPos ← |P|
13     while currentEndPos ≥ L do
14         newEndPos ← currentEndPos
15         if currentStartPos > currentEndPos − L then
16             jumpStart ← currentEndPos − L
17             newEndPos ← findSMEMendPos(P, M^rev, jumpStart)
18         if newEndPos < currentEndPos then
19             currentEndPos ← newEndPos                    // current SMEM is too short
20         else
21             currentStartPos ← findSMEMstartPos(P, M, currentEndPos)
22             foundSMEMs ← foundSMEMs ∪ {[currentStartPos, currentEndPos)}
23             if currentStartPos equals 0 then
24                 break
25             currentStartPos ← currentStartPos − 1
26             currentEndPos ← findSMEMendPos(P, M^rev, currentStartPos)
27     return foundSMEMs
```

Algorithm 3. Update the tag toehold for a partial match *P* with interval ($[s,e),R_s,R_e$), using move table *M* and sampled tag array *t*. The match *P*, corresponding to previousTagToehold, will be extended by prepending character *c*.

```
def updateTagToehold(M, t, ([s, e), R_s, R_e), c, previousTagToehold):
    c' ← M[R_e].c
    if c' equals c then
        return previousTagToehold
    else
        (e_c, R_{e,c}) ← M.walkToPreviousRun([s, e), R_s, R_e, c)
        return t[R_{e,c}]
```

RESOURCE AVAILABILITY

Lead contact

Requests for further information and resources should be directed to and will be fulfilled by the lead contact, Lore Depuydt (lore.depuydt@gmail.com).

Materials availability

This study did not generate new materials.

Data and code availability

- **Data:** The datasets supporting the results of this work are publicly available at https://github.com/biointec/tagger/tree/data/BacterialGenomes, https://github.com/biointec/tagger/tree/data/SILVA, and https://github.com/PacificBiosciences/DevNet/wiki/Microbial-Multiplexing:-PacBio-Sequel-System,-Chemistry—v3.0,-Analysis—SMRT-Link-v6.0.0.
- **Code:** Source code is available at https://github.com/biointec/tagger and archived on Zenodo at https://doi.org/10.5281/zenodo.17546758 under the GNU AGPL v3.0 license.
- **Other items:** Any additional information required to reanalyse the data reported in this article is available from the lead contact upon request.

ACKNOWLEDGMENTS

Lore Depuydt was funded by a PhD Fellowship FR (1117322N), Research Foundation - Flanders (FWO). Travis Gagie was funded by NSERC grant 07185-2020. Ben Langmead and Omar Ahmed were funded by NSF grant DBI-2029552 and NIH grant R01HG011392.

AUTHOR CONTRIBUTIONS

All authors conceptualized the approach. L.D. implemented the algorithms. L.D. and O.A. performed all benchmarks. J.F., B.L., and T.G. supervised the study. All authors wrote and approved the article.

DECLARATION OF INTERESTS

The authors declare no competing interests.

STAR★METHODS

Detailed methods are provided in the online version of this paper and include the following:

SUPPLEMENTAL INFORMATION

Supplemental information can be found online at https://doi.org/10.1016/j.isci.2025.114029.

Received: August 27, 2025
Revised: October 19, 2025
Accepted: November 10, 2025
Published: November 14, 2025

REFERENCES

1. Miller, S., and Chiu, C. (2021). The Role of Metagenomics and Next-Generation Sequencing in Infectious Disease Diagnosis. Clin. Chem. *68*, 115–124. https://doi.org/10.1093/clinchem/hvab173.
2. Nagata, N., Nishijima, S., Kojima, Y., Hisada, Y., Imbe, K., Miyoshi-Akiyama, T., Suda, W., Kimura, M., Aoki, R., Sekine, K., et al. (2022). Metagenomic Identification of Microbial Signatures Predicting Pancreatic Cancer From a Multinational Study. Gastroenterology *163*, 222–238. https://doi.org/10.1053/j.gastro.2022.03.054.
3. Pillay, S., Calderón-Franco, D., Urhan, A., and Abeel, T. (2022). Metagenomic-based surveillance systems for antibiotic resistance in non-clinical settings. Front. Microbiol. *13*, 1066995. https://doi.org/10.3389/fmicb.2022.1066995.
4. Taş, N., de Jong, A.E., Li, Y., Trubl, G., Xue, Y., and Dove, N.C. (2021). Metagenomic tools in microbial ecology research. Curr. Opin. Biotechnol. *67*, 184–191. https://doi.org/10.1016/j.copbio.2021.01.019.
5. Nwachukwu, B.C., and Babalola, O.O. (2022). Metagenomics: A Tool for Exploring Key Microbiome With the Potentials for Improving Sustainable Agriculture. Front. Sustain. Food Syst. *6*, 886987. https://doi.org/10.3389/fsufs.2022.886987.
6. Wood, D.E., Lu, J., and Langmead, B. (2019). Improved metagenomic analysis with Kraken 2. Genome Biol. *20*, 257. https://doi.org/10.1186/s13059-019-1891-0.
7. Kim, D., Song, L., Breitwieser, F.P., and Salzberg, S.L. (2016). Centrifuge: rapid and sensitive classification of metagenomic sequences. Genome Res. *26*, 1721–1729. https://doi.org/10.1101/gr.210641.116.
8. Blanco-Míguez, A., Beghini, F., Cumbo, F., McIver, L.J., Thompson, K.N., Zolfo, M., Manghi, P., Dubois, L., Huang, K.D., Thomas, A.M., et al. (2023). Extending and improving metagenomic taxonomic profiling with uncharacterized species using MetaPhlAn 4. Nat. Biotechnol. *41*, 1633–1644. https://doi.org/10.1038/s41587-023-01688-w.

9. Segata, N., Waldron, L., Ballarini, A., Narasimhan, V., Jousson, O., and Huttenhower, C. (2012). Metagenomic microbial community profiling using unique clade-specific marker genes. Nat. Methods *9*, 811–814. https://doi.org/10.1038/nmeth.2066.

10. Ferragina, P., and Manzini, G. (2000). Opportunistic data structures with applications. In Proceedings of the 41st Annual Symposium on Foundations of Computer Science (IEEE), pp. 390–398. https://doi.org/10.1109/SFCS.2000.892127.

11. Burrows, M., and Wheeler, D. (1994). A Block-Sorting Lossless Data Compression Algorithm. In Research Report 124 Digital Equipment Corporation Systems Research Center.

12. Nasko, D.J., Koren, S., Phillippy, A.M., and Treangen, T.J. (2018). RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. Genome Biol. *19*, 165. https://doi.org/10.1186/s13059-018-1554-6.

13. Bonnie, J.K., Ahmed, O.Y., and Langmead, B. (2024). DandD: Efficient measurement of sequence growth and similarity. iScience *27*, 109054. https://doi.org/10.1016/j.isci.2024.109054.

14. Mäkinen, V., and Navarro, G. (2005). Succinct suffix arrays based on run-length encoding. In Combinatorial Pattern Matching (Springer Berlin Heidelberg), pp. 45–56. https://doi.org/10.1007/11496656_5.

15. Gagie, T., Navarro, G., and Prezza, N. (2018). Optimal-Time Text Indexing in BWT-runs Bounded Space. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018) (SIAM), pp. 1459–1477. https://doi.org/10.1137/1.9781611975031.96.

16. Gagie, T., Navarro, G., and Prezza, N. (2020). Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. J. ACM *67*, 1–54. https://doi.org/10.1145/3375890.

17. Nishimoto, T., and Tabei, Y. (2021). Optimal-Time Queries on BWT-Runs Compressed Indexes. In 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021) vol. 198 of LIPIcs (Schloss Dagstuhl – Leibniz-Zentrum für Informatik), pp. 101:1–101:15. https://doi.org/10.4230/LIPICS.ICALP.2021.101.

18. Rossi, M., Oliva, M., Langmead, B., Gagie, T., and Boucher, C. (2022). MONI: A Pangenomic Index for Finding Maximal Exact Matches. J. Comput. Biol. *29*, 169–187. https://doi.org/10.1089/cmb.2021.0290.

19. Boucher, C., Gagie, T., Tomohiro, I., Köppl, D., Langmead, B., Manzini, G., Navarro, G., Pacheco, A., and Rossi, M. (2021). PHONI: Streamed Matching Statistics with Multi-Genome References. Proc Data Compress Conf. *2021*, 193–202. https://doi.org/10.1109/DCC50243.2021.00027.

20. Ahmed, O., Rossi, M., Kovaka, S., Schatz, M.C., Gagie, T., Boucher, C., and Langmead, B. (2021). Pan-genomic matching statistics for targeted nanopore sequencing. iScience *24*, 102696. https://doi.org/10.1016/j.isci.2021.102696.

21. Zakeri, M., Brown, N.K., Ahmed, O.Y., Gagie, T., and Langmead, B. (2024). Movi: A fast and cache-efficient full-text pangenome index. iScience *27*, 111464. https://doi.org/10.1016/j.isci.2024.111464.

22. Depuydt, L., Renders, L., Van de Vyver, S., Veys, L., Gagie, T., and Fostier, J. (2024). b-move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index. In 24th International Workshop on Algorithms in Bioinformatics (WABI 2024) vol. 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, S.P. Pissis and W.K. Sung, eds. (Schloss Dagstuhl – Leibniz-Zentrum für Informatik), pp. 10:1–10:18. https://doi.org/10.4230/LIPIcs.WABI.2024.10.

23. Depuydt, L., Renders, L., Van de Vyver, S., Veys, L., Gagie, T., and Fostier, J. (2025). b-move: faster lossless approximate pattern matching in a run-length compressed index. Algorithms Mol. Biol. *20*, 15. https://doi.org/10.1186/s13015-025-00281-x.

24. Ahmed, O.Y., Rossi, M., Gagie, T., Boucher, C., and Langmead, B. (2023). SPUMONI 2: improved classification using a pangenome index of minimizer digests. Genome Biol. *24*, 122. https://doi.org/10.1186/s13059-023-02958-1.

25. Song, L., and Langmead, B. (2024). Centrifuger: lossless compression of microbial genomes for efficient and accurate metagenomic sequence classification. Genome Biol. *25*, 106. https://doi.org/10.1186/s13059-024-03244-4.

26. Ahmed, O.Y., Boucher, C., and Langmead, B. (2025). Robust 16S rRNA classification based on a compressed LCA index. Genome Res. gr. 279846.124. https://doi.org/10.1101/gr.279846.124.

27. Ahmed, O., Rossi, M., Boucher, C., and Langmead, B. (2023). Efficient taxa identification using a pangenome index. Genome Res. *33*, 1069–1077. https://doi.org/10.1101/gr.277642.123.

28. Li, H. (2012). Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics *28*, 1838–1844. https://doi.org/10.1093/bioinformatics/bts280.

29. Gagie, T. (2024). How to Find Long Maximal Exact Matches and Ignore Short Ones. In Developments in Language Theory - 28th International Conference, DLT 2024, Göttingen, Germany, August 12-16, 2024, Proceedings vol. 14791 of *Lecture Notes in Computer Science*, J.D. Day and F. Manea, eds. (Springera), pp. 131–140. https://doi.org/10.1007/978-3-031-66159-4_10.

30. Li, H. (2024). BWT construction and search at the terabase scale. Bioinformatics *40*, btae717. https://doi.org/10.1093/bioinformatics/btae717.

31. Gagie, T. (2024). Tag arrays. Preprint at arXivb. https://doi.org/10.48550/ARXIV.2411.15291.

32. Ono, Y., Asai, K., and Hamada, M. (2021). PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. Bioinformatics *37*, 589–595. https://doi.org/10.1093/bioinformatics/btaa835.

33. Huang, W., Li, L., Myers, J.R., and Marth, G.T. (2012). ART: a next-generation sequencing read simulator. Bioinformatics *28*, 593–594. https://doi.org/10.1093/bioinformatics/btr708.

34. Jacobsen, A., Hendriksen, R.S., Aaresturp, F.M., Ussery, D.W., and Friis, C. (2011). The Salmonella enterica Pan-genome. Microb. Ecol. *62*, 487–504. https://doi.org/10.1007/s00248-011-9880-1.

35. Vicedomini, R., Quince, C., Darling, A.E., and Chikhi, R. (2021). Strainberry: automated strain separation in low-complexity metagenomes using long reads. Nat. Commun. *12*, 4485. https://doi.org/10.1038/s41467-021-24515-9.

36. Pacific Biosciences (2021). lima: Pacbio secondary analysis tool on bioconda (via pbbioconda). GitHub. URL: https://github.com/PacificBiosciences/pbbioconda

37. Devanga Ragupathi, N.K., Muthuirulandi Sethuvel, D.P., Inbanathan, F.Y., and Veeraraghavan, B. (2018). Accurate differentiation of Escherichia coli and Shigella serogroups: challenges and strategies. New Microbes New Infect. *21*, 58–62. https://doi.org/10.1016/j.nmni.2017.09.003.

38. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., and Mun, T. (2019). Prefix-free parsing for building big BWTs. Algorithms Mol. Biol. *14*, 13–15. https://doi.org/10.1186/S13015-019-0148-5.

39. Díaz-Domínguez, D., and Navarro, G. (2023). Efficient construction of the BWT for repetitive text using string compression. Inf. Comput. *294*, 105088. https://doi.org/10.1016/j.ic.2023.105088.

40. Renders, L., Depuydt, L., Rahmann, S., and Fostier, J. (2024). Lossless Approximate Pattern Matching: Automated Design of Efficient Search Schemes. J. Comput. Biol. *31*, 975–989. https://doi.org/10.1089/cmb.2024.0664.

41. Cobas, D., Gagie, T., and Navarro, G. (2021). A Fast and Small Subsampled R-Index. In 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021) vol. 191 of *Leibniz International Proceedings in Informatics (LIPIcs)* (Schloss Dagstuhl – Leibniz-Zentrum für Informatik), pp. 13:1–13:16. https://doi.org/10.4230/LIPIcs.CPM.2021.13.

42. Goga, A., Depuydt, L., Brown, N.K., Fostier, J., Gagie, T., and Navarro, G. (2024). Faster Maximal Exact Matches with Lazy LCP Evaluation. In 2024 Data Compression Conference (DCC) (IEEE), pp. 123–132. https://doi.org/10.1109/DCC58796.2024.00020.

43. Parks, D.H., Chuvochina, M., Waite, D.W., Rinke, C., Skarshewski, A., Chaumeil, P.A., and Hugenholtz, P. (2018). A standardized bacterial taxonomy based on genome phylogeny substantially revises the tree of life. Nat. Biotechnol. *36*, 996–1004. https://doi.org/10.1038/nbt.4229.

44. Parks, D.H., Chuvochina, M., Chaumeil, P.A., Rinke, C., Mussig, A.J., and Hugenholtz, P. (2020). A complete domain-to-species taxonomy for Bacteria and Archaea. Nat. Biotechnol. *38*, 1079–1086. https://doi.org/10.1038/s41587-020-0501-8.

45. Hunt, M., Lima, L., Anderson, D., Bouras, G., Hall, M., Hawkey, J., Schwengers, O., Shen, W., Lees, J.A., and Iqbal, Z. (2024). Allthebacteria – all bacterial genomes assembled, available, and searchable. Preprint at bioRxiv. https://doi.org/10.1101/2024.03.08.584059.

46. Baláž, A., Gagie, T., Goga, A., Heumos, S., Navarro, G., Petescia, A., and Sirén, J. (2024). Wheeler maps. In LATIN 2024: Theoretical Informatics (Springer Nature), pp. 178–192. https://doi.org/10.1007/978-3-031-55598-5_12.

STAR★METHODS

KEY RESOURCES TABLE

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Deposited data		
Pan-genome dataset for eight bacterial species (RefSeq genomes)	NCBI RefSeq	Genome identifiers available on GitHub
16S rRNA gene dataset	SILVA SSU NR99 (version 138.1)	Processed accession list available on GitHub
Microbial multiplexed sequencing dataset (10-plex, PacBio Sequel System)	Pacific Biosciences DevNet	Data available at PacBio DevNet and PacBio Cloud Download
Software and algorithms		
tagger	This paper	https://github.com/biointec/tagger; https://doi.org/10.5281/zenodo.17546758
SPUMONI 2	Ahmed et al.[24]	https://github.com/oma219/spumoni
Cliffy	Ahmed et al.[26]	https://github.com/oma219/cliffy
Kraken 2	Wood et al.[6]	https://github.com/DerrickWood/kraken2
PBSIM2	Ono et al.[32]	https://github.com/yukiteruono/pbsim2
ART	Huang et al.[33]	https://www.niehs.nih.gov/research/resources/software/biostatistics/art
lima	GitHub	https://github.com/PacificBiosciences/pbbioconda

METHOD DETAILS

Preliminaries

In this paper, the search text T, with length $n = |T|$, represents the concatenation of all candidate sequences for classification. T is expected to be a string over the alphabet $\Sigma = \{A, C, G, T\}$. If a candidate sequence contains a non-ACGT character, it is replaced with the dummy character 'X', which is never matched to ensure that no SMEMs span such characters. Additionally, the same dummy character 'X' is placed between each candidate sequence to prevent SMEMs from spanning multiple sequences. The concatenated text T is always terminated with the sentinel character '$', which is lexicographically smaller than any other character in the extended alphabet $\Sigma' = \{\$, A, C, G, T, X\}$.

Furthermore, we use the following notation. Array and string indexing start at 0. A substring within a text T is defined by a half-open interval $[i,j)$ over T, where $0 \leq i \leq j \leq n$. An empty interval is denoted by $[i,i)$. Intervals over other data structures, like the BWT, will also be presented as half-open intervals.

Backward and forward character matching with move tables

Finding *full* SMEMs of a read relative to the candidate dataset requires supporting backward and forward character extensions in the index. To achieve this, we leverage the cache-efficient principles of the move structure.[17] Our implementation follows the approach introduced for b-move. In this section, we briefly recapitulate the elements necessary for SMEM-finding in the context of this paper. For a more detailed overview, including pseudo-code, we refer the reader to the b-move papers.[22,23]

In the move structure, a move table facilitates fast LF-operations by mapping a character in the Last column of the lexicographically ordered rotations of T (i.e., the BWT) to its corresponding position in the First column. The LF move table M for a search text T consists of r rows, each representing a run in the BWT. Each row consists of four values: the run character c, the BWT start index p of the run, the LF mapping $\pi = \text{LF}(p)$, and the run index ξ that contains BWT index π. Table 1 shows the move table M for a small conceptual example consisting of two strains: ''CTATGTC'' and ''ATATGTTGGTC''. For completeness, the corresponding FM-index is provided in Table S1.

Since all BWT positions within the same run map to consecutive positions in the First column, the move table M suffices to perform the LF operation by accessing only a single row. Specifically, the LF mapping of BWT index i, located in run j, is calculated as $i' = M[j].\pi + (i - M[j].p)$. To determine the run where i' resides, we initially check whether run $M[j].\xi$ contains index i'. If not, we fast forward through subsequent runs until the correct run j' containing index i' is located. This functionality is bundled in the function $M.\text{LF}(i,j) = (i',j')$, which returns the LF mapping of a BWT index and its corresponding run.

Using this LF functionality, we can perform backward character extensions with the move table as follows. Let the interval [*s*,*e*) over the BWT represent a pattern *P*, meaning all lexicographically sorted rotations of *T* within this interval are prefixed by *P*. The indices *s* and *e*-1 lie within runs R_s and R_e, respectively. To extend *P* to *cP*, we must identify the smallest interval $[s_c,e_c) \subseteq [s,e)$ that contains all occurrences of character *c* in the BWT within [*s*,*e*), along with their updated run indices $R_{s,c}$ and $R_{e,c}$. This can be achieved by traversing inward over the rows of the move table, starting from runs R_s and R_e. This functionality is encapsulated in the functions M.`walkToNextRun`([*s*,*e*),R_s,R_e,*c*) (returning (s_c,$R_{s,c}$)) and M.`walkToPreviousRun`([*s*,*e*),R_s,R_e,*c*) (returning (e_c,$R_{e,c}$)). Once $[s_c,e_c)$ and the corresponding run indices are found, we compute the updated BWT interval [*s*′,*e*′) for pattern *cP*, where *s*′ and *e*′-1 are located in runs R'_s and R'_e, respectively. The output is computed as $(s', R'_s) = M.\mathrm{LF}(s_c, R_{s,c})$ and $(e' - 1, R'_e) = M.\mathrm{LF}(e_c\text{-}1, R_{e,c})$. This entire process is encapsulated in the function $([s',e'), R'_s, R'_e) = M.\mathrm{addChar}([s,e), R_s, R_e, c)$.

Example

Assume we want to backward match the pattern "TGT" using the move table *M* in Table 1. We initialize the empty move interval as ([0,20),0,13), spanning all positions in the BWT and all rows in *M*. To match the last character "T", we first call M.`walkToNextRun` and M.`walkToPreviousRun` to find the first and last occurrences of "T" in the BWT along with their corresponding runs. This results in pairs (2,2) and (15,10), respectively. Performing the LF mapping on these (position, run) pairs gives (11,7) and (18,12). Thus, the move interval for "T" is ([11,19),7,12). Similarly, two subsequent calls to M.`addChar` for characters "G" and "T" yield move intervals ([8,11),5,6) and ([16,18),11,11), respectively.

The overview above describes backward character matching using move table *M*, enabling the extension of *P* to *cP*. However, our full SMEM-finding algorithm also requires support for forward matching, i.e., extending *P* to *Pc*. To facilitate this symmetrical functionality, we store a second move table, M^{rev}, which represents the reverse of the search text, T^{rev}, in r^{rev} rows. Using M^{rev}, pattern *P* can be extended to *Pc* by computing $([s^{\mathrm{rev}'}, e^{\mathrm{rev}'}), R_s^{\mathrm{rev}'}, R_e^{\mathrm{rev}'}) = M^{\mathrm{rev}}.\mathrm{addChar}([s^{\mathrm{rev}}, e^{\mathrm{rev}}), R_s^{\mathrm{rev}}, R_e^{\mathrm{rev}}, c)$. Table 2 shows the move table M^{rev}, corresponding to our two example strains. For completeness, the FM-index for the reverse search text is provided in Table S2.

Theoretically, function `addChar` has a worst-case time complexity of $O(r)$ or $O(r^{\mathrm{rev}})$. In practice, however, the number of steps in the move structure is very small, as demonstrated by b-move's results,[22,23] and the linear, cache-friendly memory access patterns make character extensions highly efficient. Consequently, these lossless run-length compressed move tables serve as a robust foundation for developing our SMEM-finding and metagenomic classification implementation.

Finding SMEMs of length at least *L*

Before we can classify a read, we first find all of its *full* super-maximal exact matches (SMEMs) relative to the concatenation of all candidate sequences. Formally, a non-empty substring *P*[*i*,*j*) of a pattern *P* is an SMEM if it satisfies the following conditions: $P[i,j) \in T$, $P[i\text{-}1,j) \notin T$ (or $i = 0$), and $P[i,j\text{+}1) \notin T$ (or $j = |P|$). While some sources refer to these as MEMs,[29] the term MEM is sometimes also used to describe a match that cannot be extended at a specific position in *T* but may still be extendable at other positions. For clarity, we continue using the term SMEMs.

To identify all SMEMs between a pattern *P* and search text *T*, one can use the forward-backward search algorithm originally proposed by Li.[28] However, a substantial amount of computational time is spent identifying relatively short SMEMs between *P* and *T*. This issue is further exacerbated by the increasing adoption of large pan-genome references, which raise the likelihood that short substrings of *P* occur frequently in *T*. To address this, Gagie[29] proposed an updated forward-backward algorithm that enables skipping all SMEMs shorter than a customizable threshold *L*, which is also used in ropebwt3.[30] This approach allows for a more efficient focus on SMEMs that are longer than expected by chance. In this section, we describe our implementation of a customized version of this parametrized forward-backward algorithm, leveraging solely the two move tables *M* and M^{rev} introduced above.

Auxiliary Algorithm 1 and Algorithm 2 outline our approach to finding all SMEMs of length at least *L*. On line 12, we start at the end of pattern *P*. If the condition on line 15 fails (see further), the algorithm skips *L* characters ahead (in our case, backward) on line 16, which we refer to as a *jump start*. If the SMEM ending at the current *end* position is at least *L* characters long, it will bypass this jump start index. We can verify this by performing forward matching from the jump start toward the current end (line 17), using the auxiliary function in Algorithm 1 (which in turn calls the `addChar` functionality). If we are unable to reach the current end position, the SMEM is shorter than *L*, and we skip it, starting with the new end position found during forward matching (line 19).

If, however, we do reach the correct end position, we know that an SMEM of length at least *L* ending at this position exists. We then compute the length of this SMEM by backward matching from this end position (line 21) and report the result on line 22. If the beginning of the pattern *P* has not yet been reached (line 23), we update the current end position by performing forward matching from the position immediately preceding the last found SMEM (line 26). The process then repeats. Note that if the part of the next SMEM that was found on line 26 is already of length at least *L*, the next jump start can be skipped (see line 15).

Example

We demonstrate the SMEM-finding algorithm on *P* = "CTATGTTGCTC", a recombination of the running example strains ("CTATGTC" and "ATATGTTGGTC") with a "sequencing error" at the third-to-last position. For the sake of the example, we search for all SMEMs of length at least $L = 3$. Figure 2 shows the required character extensions, their order, and direction.

The algorithm always starts with a jump start at the end of *P*. Due to the sequencing error at position 8, the first two jump starts (line 17) fail to match the 3-mers "CTC" and "GCT" in the search text. The third jump start succeeds, after which we locate the SMEM start position using a forward search (line 21). Subsequently, no further jump starts are needed, as the overlap with the next (and last) SMEM exceeds *L*-1. This yields two SMEMs: *P*[1,8) and *P*[0,6), effectively skipping shorter SMEMs at the pattern's end.

Note that when a jump start succeeds, it performs some redundant work, as the subsequent call to `findSMEMstartPos` repeats the same character extensions. However, in practice, this redundancy is outweighed by the computational savings achieved through skipping short SMEMs with failed jump starts.

Lookup optimization

In Algorithms 1 and 2, a significant amount of time is spent matching the first characters of a partial match against the dense part of the search tree. To mitigate this, a lookup table can store forward and backward intervals for all possible k-mers, where k is set to 10 by default. Both intervals can be precomputed simultaneously using b-move's bidirectional matching functionality.[22,23]

For the jump start on line 17, the match length is required to be L, so the lookup table can be applied assuming that $L \geq k$. If the lookup fails, we lose precision over the new currentEndPos value, but the performance gains outweigh this limitation in practice. Similarly, on line 21, we know the SMEM length is at least $L \geq k$, allowing use of the lookup table. Only the call to `findSMEMendPos` on line 26 lacks guarantees on match length, so the lookup table cannot be applied there.

Including the lookup table increases the index's theoretical space complexity from $O(r+r^{\text{rev}})$ to $O(r+r^{\text{rev}}+4^k)$, assuming $L \geq k$. However, in practical use cases, the lookup table's space usage remains relatively small compared to the other index components.

Assigning a class tag to each SMEM

Storing a sampled tag array

To perform metagenomic read classification, we assign a metagenomic class identifier to each SMEM. To do this, we store a sampled tag array alongside the two move tables. A tag array "tags" each position in the BWT with metadata,[31] such as positions in a pan-genome graph[46] or, in our case, metagenomic class identifiers. In our tag array, each position in the BWT is tagged with its corresponding metagenomic class identifier. Realistic datasets typically contain multiple species within each class, so sequences from the same class tend to cluster together in the BWT. While this contextual locality could be used to directly run-length compress the tag array,[31] we opt to sample the tag array at the BWT run end positions. Sampling at the BWT run ends retains $O(r+r^{\text{rev}})$ memory complexity and proves more memory-efficient for datasets with many metagenomic classes, without negatively impacting classification performance.

Example. In our example, let the first strain ("CTATGTC") correspond to metagenomic class 0 and the second strain ("ATATGTTGGTC") to class 1. Table 1 shows the corresponding sampled tag array t. Dashes indicate positions that can contain any value, as they are never queried (they correspond to prefixes starting with "$" or "X"). For completeness, the unsampled tag array is provided in Table S1. Note that this example is too small to demonstrate contextual locality, as it contains only one sequence per class.

Updating the tag toehold

To assign a class tag to each SMEM, we track *one* tag value while matching the SMEM to the index. We call this value the "tag toehold", analogous to the suffix array toehold kept in the r-index.[15,16] This tag toehold always represents the class tag of the *last* index within the BWT interval of the current partial match. When the full SMEM is identified, it is assigned that tag of its last occurrence in the BWT, even if it appears in other metagenomic classes. Since we search for SMEMs of length at least L (i.e., with high specificity) and expect contextual locality in the tag array, this single tag is assumed to be representative in most cases. This approach avoids the computationally expensive process of querying the class tags for *all* occurrences of an SMEM, which can be numerous in pan-genomic contexts.

To maintain the tag toehold while backward matching an SMEM (line 21, Alg. 2), it must be updated each time *before* a successful `addChar` operation is performed in `findSMEMstartPos` (Alg. 1). Algorithm 3 outlines this process. The update starts with the BWT interval $[s,e)$ of the current partial match P, the run indices R_s and R_e corresponding to BWT indices s and e-1, the character c about to be prepended to P, and the tag toehold from the previous iteration. For the first iteration, the tag toehold is initialized to $t[r\text{-}1]$. The algorithm checks if the character c' of BWT run R_e matches c on line 3. If so, the last occurrence of P's BWT interval also corresponds to the last occurrence of cP's BWT interval, and the previous tag toehold remains valid (line 4). Otherwise, the largest BWT index $(e_c\text{-}1) \in [s,e)$ with $\text{BWT}[e_c\text{-}1] = c$ and its run index $R_{e,c}$ are located using the `walkToPreviousRun` function (line 6). The updated tag toehold is then set to the last tag of run $R_{e,c}$, which is sampled at $t[R_{e,c}]$. If no such e_c index exists, the end of the SMEM has been found and the toehold must no longer be updated.

Example. We illustrate the toehold updating process for SMEM $P[0,6)$ = "CTATGT", found for pattern P = "CTATGTTGCTC" above. All operations can be verified using Tables 2 and S1. Starting with a lookup table where k = 3, the interval for $P[3,6)$ = "TGT" is ([16,18),11,11), with tag toehold 1. The run character $M[11].c$ = "A" matches the next character of P, so the toehold remains 1. After extending with "A", the interval for $P[2,6)$ = "ATGT" becomes ([2,4),2,2). Again, $M[2].c$ = "T" matches the next character in P, so the toehold remains 1. Next, the interval for $P[1,6)$ = "TATGT" is ([11,13),7,8). Here, $M[8].c$ = "A" does not match $P[0]$ = "C", so we compute M.`walkToPreviousRun`([11,13),7,8,"C")=(12,7), where 12 is the exclusive upper bound, yielding final toehold $t[7] = 0$.

Consensus classification

After processing a pattern and identifying its SMEMs with tag toeholds, we assign a consensus metagenomic class to the read. This is done by tracking the accumulated SMEM length for each class as evidence. To avoid bias, overlapping SMEM regions from the same class are counted only once. This ensures that multiple shorter, overlapping SMEMs do not outweigh a single long, high-evidence SMEM. The class with the highest evidence from non-overlapping SMEMs is selected as the consensus tag. Ties are broken randomly.

For read data, the process is repeated for the reverse complement. The final consensus tag is based on the orientation (forward or reverse complement) with the most evidence across all classes.

QUANTIFICATION AND STATISTICAL ANALYSIS

All analyses were computational. Runtimes represent the median of 20 runs and exclude index loading times. Accuracy, sensitivity, specificity, and precision were computed directly from simulated or demultiplexed ground-truth read labels as described in the Results section. All reported performance and accuracy values were measured directly and are summarized in the figures and tables. No biological experiments or inferential statistical tests were performed, as all reported values are derived from direct computational measurements.