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

Lore Depuydt[1,*], Omar Y. Ahmed[2],
Jan Fostier[1], Ben Langmead[2], and Travis Gagie[3]

[1] Department of Information Technology - IDLab, Ghent University - imec
[2] Department of Computer Science, Johns Hopkins University
[3] Faculty of Computer Science, Dalhousie University
[*] *Corresponding author:* `Lore.Depuydt@UGent.be`

Abstract

Metagenomic read classification is a fundamental task in computational biology, yet it remains challenging due to the scale, diversity, and complexity of sequencing datasets. We propose a novel, lossless, run-length compressed index that enables efficient multi-class metagenomic classification in $O(r)$ space, based on the move structure. Our method identifies all super-maximal exact matches (SMEMs) of length at least L between a read and the reference dataset and associates each SMEM with one class identifier using a sampled tag array. A consensus algorithm then compacts these SMEMs with their class identifier into a single classification per read. We are the first to perform run-length compressed read classification based on full SMEMs instead of semi-SMEMs. We evaluate our approach on both long and short reads in two conceptually distinct datasets: a large bacterial pan-genome with few metagenomic classes and a smaller 16S rRNA gene database spanning thousands of genera or classes. Our method consistently outperforms SPUMONI 2 in accuracy and runtime, with only a modest memory overhead. Compared to Cliffy, we demonstrate better memory efficiency while achieving superior accuracy on the simpler dataset and comparable performance on the more complex one. Overall, our implementation carefully balances accuracy, runtime, and memory usage, offering a versatile solution for metagenomic classification across diverse datasets. The open-source C++11 implementation is available at `https://github.com/biointec/tagger` under the AGPL-3.0 license.

1 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. Lossy tools such as Kraken 2 and MetaPhlAn lose specificity as database size increases [12], while lossless tools such as Centrifuge experience index growth proportional to the sequence content in the reference, which becomes impractical even for high-RAM workstations.

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 in the run-length FM-index (RLFM-index) [13] and the r-index [14, 15], both achieving $O(r)$ space complexity, where r is the number of character runs in the BWT. The introduction of the move structure [16] 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 [17], PHONI [18], SPUMONI [19], Movi [20], and b-move [21, 22] have demonstrated the practical utility of run-length compressed indexes.

In metagenomic classification, tools like SPUMONI 2 [23], Centrifuger [24], and Cliffy [25] 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 [23, 17]. 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 [24], 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 [26]. 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 [25]. Notably, each of these tools perform backward search to find semi-maximal matches, but lack support for *full* SMEM-based classification. Though they demonstrate the potential of run-length compressed indexes in metagenomic classification, there is still need for further improvements in balancing space efficiency, scalability, performance, and classification accuracy.

Contributions. We propose a new, lossless, run-length compressed index that efficiently supports multi-class metagenomic classification in $O(r)$ space. Classification begins by identifying all *full* super-maximal exact matches (SMEMs) of a read relative to the reference dataset. 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. [27] (see also [28]). During SMEM identification, we use a sampled tag array of class identifiers [29] to track the identifier of *one* metagenomic class in which the SMEM occurs. To ensure $O(r)$ 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 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.

2 Methods

2.1 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 $\Sigma = \{\text{A, C, G, T}\}$. If a candidate sequence contains a non-ACGT character, it is replaced with the dummy character 'X', ensuring 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 $\Sigma' = \{\$, \text{A, C, G, T, X}\}$.

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.

2.1.1 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 [16]. 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 [21, 22].

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. The left panel of 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 Supplementary 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 $M.\texttt{LF}(i, j) = (i', j')$, 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 $M.\texttt{walkToNextRun}([s, e[, R_s, R_e, c)$ (returning $(s_c, R_{s,c})$) and $M.\texttt{walkToPreviousRun}([s, e[, R_s, R_e, c)$ (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 $(s', R'_s) = M.\texttt{LF}(s_c, R_{s,c})$ and $(e' - 1, R'_e) = M.\texttt{LF}(e_c - 1, R_{e,c})$. This entire process is encapsulated in the function $([s', e'[, R'_s, R'_e) = M.\texttt{addChar}([s, e[, R_s, R_e, c)$.

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 $([s^{\text{rev}\prime}, e^{\text{rev}\prime}[, R_s^{\text{rev}\prime}, R_e^{\text{rev}\prime}) = M^{\text{rev}}.\texttt{addChar}([s^{\text{rev}}, e^{\text{rev}}[, R_s^{\text{rev}}, R_e^{\text{rev}}, c)$. The right panel of Table 1 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 Supplementary Table S2.

Theoretically, function `addChar` has a worst-case time complexity of $O(r)$ or $O(r^{\text{rev}})$. In practice, however, the number of steps in the move structure is very small, as demonstrated by b-move's results [21, 22], and

Table 1: Left: Move table M for the search text T = "CTATGTCXATATGTTGGTC$". Right: Move table M^{rev} for the reverse search text T^{rev} = "$CTGGTTGTATAXCTGTATC". The corresponding FM-indexes are provided in Supplementary Tables S1 and S2.

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

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

the linear, cache-friendly memory access patterns make character extensions highly efficient. Consequently, these move tables serve as a robust foundation for developing our new SMEM-finding and metagenomic classification implementation.

2.2 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. An SMEM between a read and the search text is defined as an exact match that cannot be extended in either direction while still occurring in the search text [30]. Formally, a non-empty substring $P[i, j[$ of a pattern P is an SMEM if it satisfies the following conditions: $P[i, j[\in T$, $P[i-1, j[\notin T$, and $P[i, j+1[\notin T$. While some sources refer to these as MEMs [27], 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 [30]. 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 [27] proposed an updated forward-backward algorithm that enables skipping all SMEMs shorter than a customizable threshold L, which is also used in ropebwt3 [28]. 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 in Section 2.1.1.

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 from Section 2.1.1). 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

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.

```
def findSMEMendPos(P, M^rev, startPos):
    interval ← ([0, n[, 0, r^rev)
    nextPosToMatch ← startPos
    while interval is not empty and
    nextPosToMatch < |P| do
        c ← P[nextPosToMatch]
        interval ← M^rev.addChar(interval, c)
        nextPosToMatch ← nextPosToMatch + 1
    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
```

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
```

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 example strains from Section 2.1.1 ("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 1 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

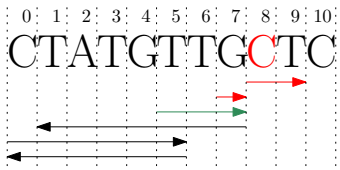

Figure 1: Visualization of the search steps in Algorithm 2 to execute `findSMEMs`$(P, M, M^{\text{rev}}, 3)$, where $P =$"CTATGTTGCTC", and M and M^{rev} correspond to the example move tables in Table 1. $P[8] = C$ represents a sequencing error. The red arrows (top two) represent failed jump start searches,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

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 [21, 22].

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.

2.3 Assigning a class tag to each SMEM

2.3.1 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 [29], such as positions in a pan-genome graph [31] or, in our case, metagenomic class identifiers. To the best of our knowledge, this work provides the first experimental implementation of a tag array. 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 [29], 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 2 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 Supplementary Table S1. Note that this example is too small to demonstrate contextual locality, as it contains only one sequence per class.

2.3.2 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 [14, 15]. 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

Table 2: The sampled tag array t, containing r metagenomic class identifiers, each corresponding to a run in the move table M (see Table 1). The full tag array is provided in Supplementary Table S1.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$t[j]$	-	1	1	0	1	1	1	0	1	0	1	1	1	-

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.

1 **def** `updateTagToehold`$(M, t, ([s, e[, R_s, R_e), c, \mathit{previousTagToehold})$**:**
2 $\quad c' \leftarrow M[R_e].c$
3 $\quad$ **if** c' **equals** c **then**
4 $\quad\quad$ **return** *previousTagToehold*
5 $\quad$ **else**
6 $\quad\quad (e_c, R_{e,c}) \leftarrow M$.`walkToPreviousRun`$([s, e[, R_s, R_e, c)$
7 $\quad\quad$ **return** $t[R_{e,c}]$

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 $t[r-1]$. 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 $(e_c - 1) \in [s, e[$ with $\mathrm{BWT}[e_c - 1] = c$ and its run index $R_{e,c}$ are located using the `walkToPreviousRun` function from Section 2.1.1 (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" in Section 2.2. All operations can be verified using Tables 1 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 $t[7] = 0$.

2.4 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.

3 Results

3.1 Setup and hardware

We compare the accuracy and performance of our metagenomic read classification implementation with SPUMONI 2 [23] and Cliffy [25], 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[1]. 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 [23]. Post-processing majority vote runtimes are excluded from benchmarks.

Cliffy, though mainly intended for taxonomic classification, also supports multi-class metagenomic classification[2]. 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. [25]. This consensus algorithm is excluded from runtime benchmarks.

All benchmarks used a single core of two 18-core Intel® Xeon® Gold 6140 CPUs (2.30 GHz) and 177 GiB of RAM. Runtimes represent the median of 20 runs and exclude index loading times.

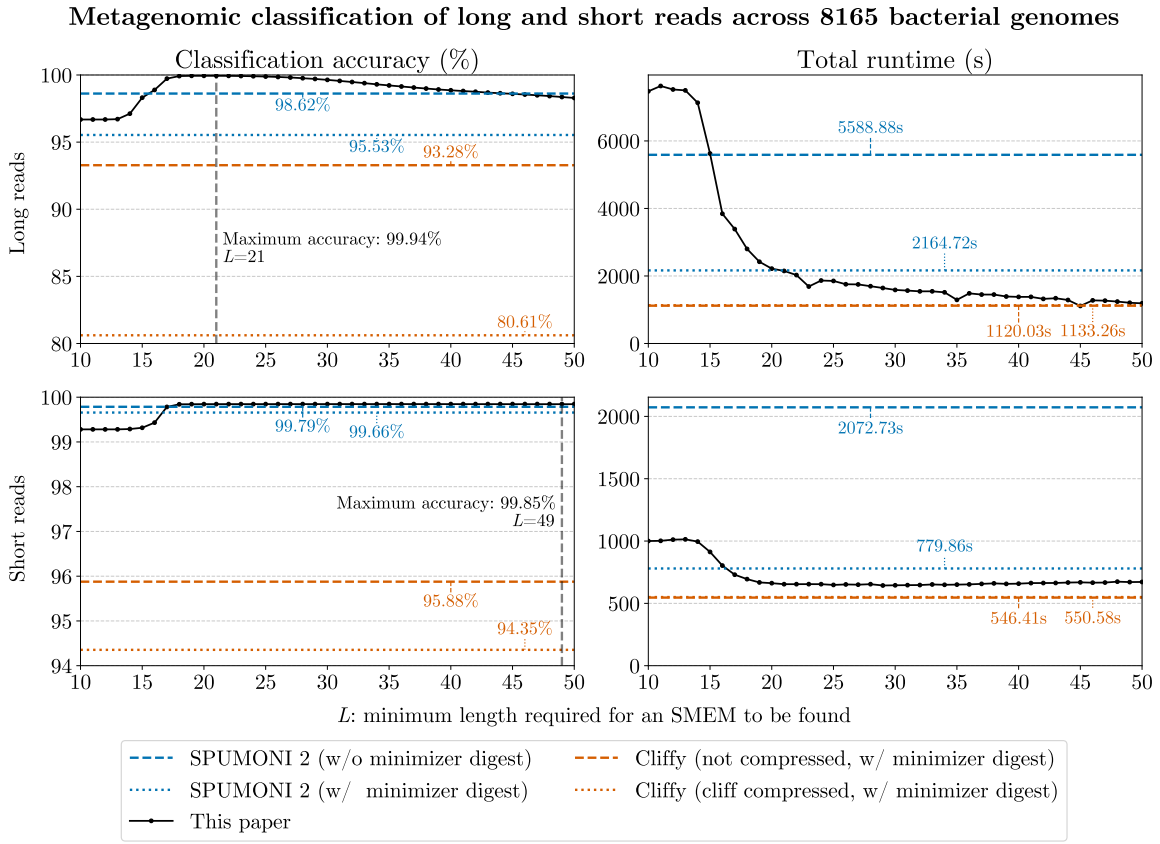

Figure 2: Analysis of metagenomic classification accuracy (left) and runtime performance (right) for matching 400 000 long reads (top) and 4 000 000 short reads (bottom) to 8 165 bacterial genomes across 8 metagenomic classes. We compare our approach for varying values of L, to SPUMONI 2 (without/with minimizer digests) and Cliffy with minimizer digests (with uncompressed/cliff compressed document array profiles).

[1] `SPUMONI v2.0.9: spumoni build/run --PML --doc-array --no-digest/--minimizer-alphabet`

[2] `cliffy v2.0.0: cliffy build --revcomp --two-pass [--minimizers] [--taxcomp --num-col 7]; cliffy run --ftab [--minimizers] [--taxcomp --num-col 7]`

3.2 Case study with 8 165 bacterial genomes

We performed a case study similar to the SPUMONI 2 multi-class classification experiment [23]. 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[3] across 8 metagenomic classes, totaling 37.4 Gbp. To assess read classification accuracy, we simulated 50 000 long reads[4] [32] (average length 5 236 bp) and 500 000 short reads[5] [33] (length 250 bp) per species.

Figure 2 compares the accuracy and runtime performance of our approach against SPUMONI 2 and Cliffy for this dataset (an extended version of the bottom left plot is shown in Supplementary 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 and Cliffy in accuracy across all configurations, as shown in the left panels. While the improvement over SPUMONI 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.

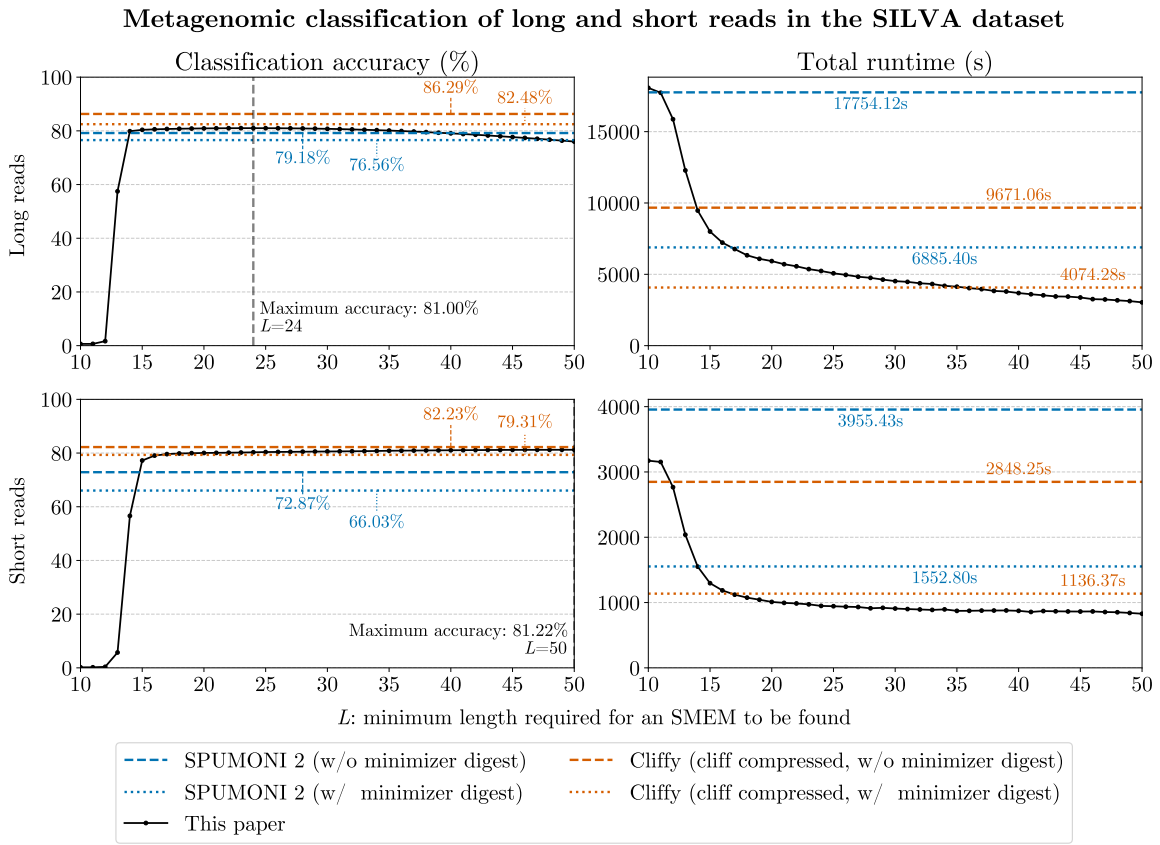

Figure 3: Analysis of metagenomic classification accuracy (left) and runtime performance (right) for matching 5 726 966 long reads (top) and 9 029 669 short reads (bottom) to 402 378 16S rRNA gene sequences spanning 9 118 metagenomic classes. We compare our approach for varying values of L, to SPUMONI 2 (without/with minimizer digests) and Cliffy with cliff compressed document array profiles (without/with minimizer digests).

[3]Details available at `https://github.com/biointec/tagger/tree/data/BacterialGenomes`
[4]`PBSIM v2.0.1: pbsim --hmm_model R94.model --accuracy-mean 0.95`
[5]`ART_Illumina v2.3.7: art_illumina --seqSys MS --len 250`

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 for 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.

3.3 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 [25]. From this collection, we selected the 402 378 16S rRNA gene sequences[6] annotated to the genus level, totaling to 586 Mbp of data spanning 9 118 genera, which serve as the metagenomic classes. Compared to the dataset in Section 3.2, 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 250 bp), aiming for equal coverage across genera.

Figure 3 compares our approach to SPUMONI 2 and Cliffy (an extended version of the bottom left plot is provided in Supplementary Figure S2). For the SILVA dataset, only Cliffy indexes with cliff compressed document array profiles are considered, as their uncompressed counterparts exceed 1 TiB of memory [25]. For classification accuracy, our approach (for optimal L) consistently outperforms SPUMONI 2 for both read types. 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 significantly outperforms SPUMONI 2 across all configurations. For this dataset, we also surpass Cliffy in runtime, except when matching long reads with $L \leq 35$, where Cliffy with minimizer digests achieves both higher accuracy and faster performance.

3.4 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 (Sections 3.2 and 3.3), and Supplementary Figure S3 covers short reads. For our approach, we fix L at 25, balancing accuracy and runtime as demonstrated above.

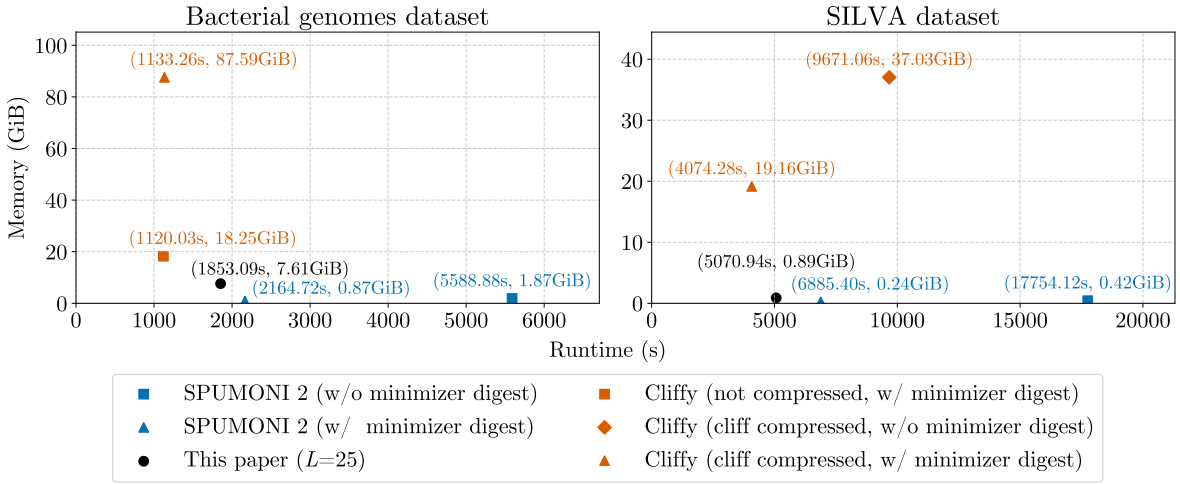

Figure 4: Memory versus runtime comparison of our approach ($L = 25$) with SPUMONI 2 and Cliffy, using all configurations the dataset allows. Shown are results for matching 400 000 long reads to an index of 8 165 bacterial genomes across 8 metagenomic classes (left); and for matching 5 726 966 long reads to an index of 402 378 16S rRNA gene sequences spanning 9 118 metagenomic classes (right).

[6]Details available at `https://github.com/biointec/tagger/tree/data/SILVA`

For the bacterial genomes dataset, where Cliffy shows significantly lower accuracy (Section 3.2), 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 within 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 (Section 3.3) 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 faster and more accurate. Both our index and SPUMONI 2's indexes fit within 1 GiB, however.

4 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 [34]. The suffix array approach is faster but memory-intensive, suitable for smaller datasets like the SILVA dataset (under 1 hour runtime, 6.2 GiB peak RAM). Prefix-free parsing is slower but uses less memory, making it ideal for larger pan-genomes like the bacterial genomes dataset (approximately 13 hours runtime, 106 GiB peak RAM, feasible on a regular workstation). We aim to improve our construction implementation based on recent advancements such as grlBWT [35] and ropebwt3 [28].

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 paper, 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 [36] 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 performance penalty, similar to subsampling the suffix array samples in the sr-index [37, 38]. However, the relative memory reduction is modest, as the majority of the index consists of the two move tables. Finally, we also support parallelization.

5 Conclusion

We introduced a novel 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, with only a modest memory overhead. While Cliffy achieves slightly higher accuracy for the complex dataset with thousands of metagenomic classes, this comes at the cost of a significant memory increase (20 to 40 times more). 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.

Acknowledgements

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.

References

[1] Miller S, Chiu C. The Role of Metagenomics and Next-Generation Sequencing in Infectious Disease Diagnosis. Clinical Chemistry. 2021 12;68(1):115-24. doi:10.1093/clinchem/hvab173.

[2] Nagata N, Nishijima S, Kojima Y, Hisada Y, Imbe K, Miyoshi-Akiyama T, et al. Metagenomic Identification of Microbial Signatures Predicting Pancreatic Cancer From a Multinational Study. Gastroenterology. 2022;163(1):222-38. doi:10.1053/j.gastro.2022.03.054.

[3] Pillay S, Calderón-Franco D, Urhan A, Abeel T. Metagenomic-based surveillance systems for antibiotic resistance in non-clinical settings. Frontiers in Microbiology. 2022;13. doi:10.3389/fmicb.2022.1066995.

[4] Taş N, de Jong AE, Li Y, Trubl G, Xue Y, Dove NC. Metagenomic tools in microbial ecology research. Current Opinion in Biotechnology. 2021;67:184-91. doi:10.1016/j.copbio.2021.01.019.

[5] Nwachukwu BC, Babalola OO. Metagenomics: A Tool for Exploring Key Microbiome With the Potentials for Improving Sustainable Agriculture. Frontiers in Sustainable Food Systems. 2022;6. doi:10.3389/fsufs.2022.886987.

[6] Wood DE, Lu J, Langmead B. Improved metagenomic analysis with Kraken 2. Genome Biology. 2019 Nov;20(1):257. doi:10.1186/s13059-019-1891-0.

[7] Kim D, Song L, Breitwieser FP, Salzberg SL. Centrifuge: rapid and sensitive classification of metagenomic sequences. Genome Research. 2016 Oct;26(12):1721–1729. doi:10.1101/gr.210641.116.

[8] Blanco-Míguez A, Beghini F, Cumbo F, McIver LJ, Thompson KN, Zolfo M, et al. Extending and improving metagenomic taxonomic profiling with uncharacterized species using MetaPhlAn 4. Nature Biotechnology. 2023 Feb;41(11):1633–1644. doi:10.1038/s41587-023-01688-w.

[9] Segata N, Waldron L, Ballarini A, Narasimhan V, Jousson O, Huttenhower C. Metagenomic microbial community profiling using unique clade-specific marker genes. Nature Methods. 2012 Jun;9(8):811–814. doi:10.1038/nmeth.2066.

[10] Ferragina P, Manzini G. Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science; 2000. p. 390-8. doi:10.1109/SFCS.2000.892127.

[11] Burrows M, Wheeler D. A Block-Sorting Lossless Data Compression Algorithm. 130 Lytton Avenue, Palo Alto, California 94301: Digital Equipment Corporation Systems Research Center; 1994. 124.

[12] Nasko DJ, Koren S, Phillippy AM, Treangen TJ. RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. Genome Biology. 2018 Oct;19(1). doi:10.1186/s13059-018-1554-6.

[13] Mäkinen V, Navarro G. Succinct Suffix Arrays Based on Run-Length Encoding. In: Combinatorial Pattern Matching. Berlin, Heidelberg: Springer Berlin Heidelberg; 2005. p. 45-56. doi:10.1007/11496656_5.

[14] Gagie T, Navarro G, Prezza N. Optimal-Time Text Indexing in BWT-runs Bounded Space. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018. SIAM; 2018. p. 1459-77. doi:10.1137/1.9781611975031.96.

[15] Gagie T, Navarro G, Prezza N. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. J ACM. 2020 January;67(1). doi:10.1145/3375890.

[16] Nishimoto T, Tabei Y. Optimal-Time Queries on BWT-Runs Compressed Indexes. In: 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference). vol. 198 of LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2021. p. 101:1-101:15. doi:10.4230/LIPICS.ICALP.2021.101.

[17] Rossi M, Oliva M, Langmead B, Gagie T, Boucher C. MONI: A Pangenomic Index for Finding Maximal Exact Matches. Journal of Computational Biology. 2022;29(2):169-87. PMID: 35041495. doi:10.1089/cmb.2021.0290.

[18] Boucher C, Gagie T, Tomohiro I, Köppl D, Langmead B, Manzini G, et al. PHONI: Streamed Matching Statistics with Multi-Genome References. In: 2021 Data Compression Conference (DCC); 2021. p. 193-202. doi:10.1109/DCC50243.2021.00027.

[19] Ahmed O, Rossi M, Kovaka S, Schatz MC, Gagie T, Boucher C, et al. Pan-genomic matching statistics for targeted nanopore sequencing. iScience. 2021;24(6):102696. doi:10.1016/j.isci.2021.102696.

[20] Zakeri M, Brown NK, Ahmed OY, Gagie T, Langmead B. Movi: A fast and cache-efficient full-text pangenome index. iScience. 2024 Dec;27(12). doi:10.1016/j.isci.2024.111464.

[21] Depuydt L, Renders L, Van de Vyver S, Veys L, Gagie T, Fostier J. b-move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index. In: Pissis SP, Sung WK, editors. 24th International Workshop on Algorithms in Bioinformatics (WABI 2024). vol. 312 of Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2024. p. 10:1-10:18. doi:10.4230/LIPIcs.WABI.2024.10.

[22] Depuydt L, Renders L, de Vyver SV, Veys L, Gagie T, Fostier J. b-move: Faster Lossless Approximate Pattern Matching in a Run-Length Compressed Index. Research Square. 2024 Nov. doi:10.21203/rs.3.rs-5367343/v1.

[23] Ahmed OY, Rossi M, Gagie T, Boucher C, Langmead B. SPUMONI 2: improved classification using a pangenome index of minimizer digests. Genome Biology. 2023 May;24(1):122. doi:10.1186/s13059-023-02958-1.

[24] Song L, Langmead B. Centrifuger: lossless compression of microbial genomes for efficient and accurate metagenomic sequence classification. Genome Biology. 2024 Apr;25(1). doi:10.1186/s13059-024-03244-4.

[25] Ahmed O, Boucher C, Langmead B. Cliffy: robust 16S rRNA classification based on a compressed LCA index. bioRxiv. 2024 May. doi:10.1101/2024.05.25.595899.

[26] Ahmed O, Rossi M, Boucher C, Langmead B. Efficient taxa identification using a pangenome index. Genome Research. 2023 May. doi:10.1101/gr.277642.123.

[27] Gagie T. How to Find Long Maximal Exact Matches and Ignore Short Ones. In: Day JD, Manea F, editors. Developments in Language Theory - 28th International Conference, DLT 2024, Göttingen, Germany, August 12-16, 2024, Proceedings. vol. 14791 of Lecture Notes in Computer Science. Springer; 2024. p. 131-40. doi:10.1007/978-3-031-66159-4_10.

[28] Li H. BWT construction and search at the terabase scale. Bioinformatics. 2024 11;40(12):btae717. doi:10.1093/bioinformatics/btae717.

[29] Gagie T. Tag arrays. CoRR. 2024;abs/2411.15291. doi:10.48550/ARXIV.2411.15291.

[30] Li H. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics. 2012 05;28(14):1838-44. doi:10.1093/bioinformatics/bts280.

[31] Baláž A, Gagie T, Goga A, Heumos S, Navarro G, Petescia A, et al. Wheeler Maps. In: LATIN 2024: Theoretical Informatics. Cham: Springer Nature Switzerland; 2024. p. 178-92. doi:10.1007/978-3-031-55598-5_12.

[32] Ono Y, Asai K, Hamada M. PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. Bioinformatics. 2020 09;37(5):589-95. doi:10.1093/bioinformatics/btaa835.

[33] Huang W, Li L, Myers JR, Marth GT. ART: a next-generation sequencing read simulator. Bioinformatics. 2011 12;28(4):593-4. doi:10.1093/bioinformatics/btr708.

[34] Boucher C, Gagie T, Kuhnle A, Langmead B, Manzini G, Mun T. Prefix-free parsing for building big BWTs. Algorithms Mol Biol. 2019;14(1):13:1-13:15. doi:10.1186/S13015-019-0148-5.

[35] Díaz-Domínguez D, Navarro G. Efficient construction of the BWT for repetitive text using string compression. Information and Computation. 2023;294:105088. doi:10.1016/j.ic.2023.105088.

[36] Renders L, Depuydt L, Rahmann S, Fostier J. Lossless Approximate Pattern Matching: Automated Design of Efficient Search Schemes. Journal of Computational Biology. 2024. PMID: 39344875. doi:10.1089/cmb.2024.0664.

[37] Cobas D, Gagie T, Navarro G. 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). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2021. p. 13:1-13:16. doi:10.4230/LIPIcs.CPM.2021.13.

[38] Goga A, Depuydt L, Brown NK, Fostier J, Gagie T, Navarro G. Faster Maximal Exact Matches with Lazy LCP Evaluation. In: 2024 Data Compression Conference (DCC); 2024. p. 123-32. doi:10.1109/DCC58796.2024.00020.

Table S1: FM-index for the search text T = "CTATGTCXATATGTTGGTC$" showing its tag array, Burrows-Wheeler transform BWT (Last column), LF mapping, and suffixes (with the First column represented by the suffixes' first characters). BWT runs are separated by horizontal lines.

i	T	tags	BWT	LF	$T_{\mathrm{SA}[i]}$
0	C	-	C	4	$
1	T	1	X	19	ATATGTTGGTC$
2	A	0	T	11	ATGTCXATATGTTGGTC$
3	T	1	T	12	ATGTTGGTC$
4	G	1	T	13	C$
5	T	0	$	0	CTATGTCXATATGTTGGTC$
6	C	0	T	14	CXATATGTTGGTC$
7	X	1	T	15	GGTC$
8	A	1	G	7	GTC$
9	T	0	T	16	GTCXATATGTTGGTC$
10	A	1	T	17	GTTGGTC$
11	T	0	C	5	TATGTCXATATGTTGGTC$
12	G	1	A	1	TATGTTGGTC$
13	T	1	G	8	TC$
14	T	0	G	9	TCXATATGTTGGTC$
15	G	1	T	18	TGGTC$
16	G	0	A	2	TGTCXATATGTTGGTC$
17	T	1	A	3	TGTTGGTC$
18	C	1	G	10	TTGGTC$
19	$	-	C	6	XATATGTTGGTC

Table S2: FM-index for the reverse search text T^{rev} = "$CTGGTTGTATAXCTGTATC" showing its Burrows-Wheeler transform $\mathrm{BWT}^{\mathrm{rev}}$, LF mapping $\mathrm{LF}^{\mathrm{rev}}$, and suffixes. BWT runs are separated by horizontal lines.

i	T^{rev}	$\mathrm{BWT}^{\mathrm{rev}}$	$\mathrm{LF}^{\mathrm{rev}}$	$T^{\mathrm{rev}}_{\mathrm{SA}^{\mathrm{rev}}[i]}$
0	$	C	4	$CTGGTTGTATAXCTGTATC
1	C	T	11	ATAXCTGTATC
2	T	T	12	ATC
3	G	T	13	AXCTGTATC
4	G	T	14	C
5	T	$	0	CTGGTTGTATAXCTGTATC
6	T	X	19	CTGTATC
7	G	T	15	GGTTGTATAXCTGTATC
8	T	T	16	GTATAXCTGTATC
9	A	T	17	GTATC
10	T	G	7	GTTGTATAXCTGTATC
11	A	G	8	TATAXCTGTATC
12	X	G	9	TATC
13	C	A	1	TAXCTGTATC
14	T	A	2	TC
15	G	C	5	TGGTTGTATAXCTGTATC
16	T	T	18	TGTATAXCTGTATC
17	A	C	6	TGTATC
18	T	G	10	TTGTATAXCTGTATC
19	C	A	3	XCTGTATC

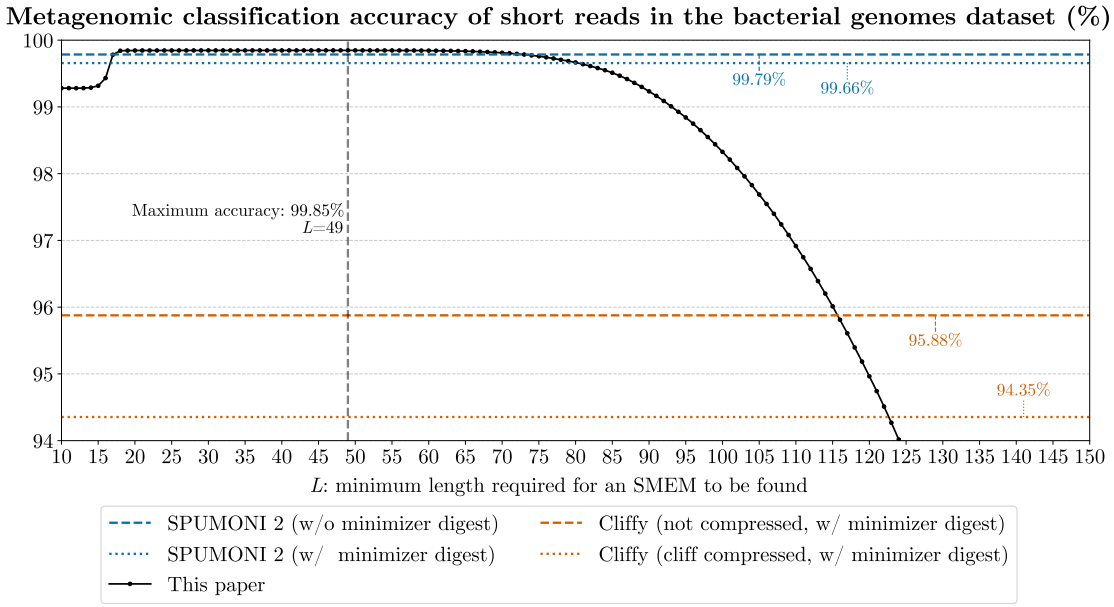

Figure S1: Analysis of metagenomic classification accuracy for matching 4 000 000 short reads to an index of 8 165 bacterial genomes across 8 metagenomic classes. We compare our approach for varying values of L, to SPUMONI 2 (without and with minimizer digests) and Cliffy with minimizer digests (with uncompressed and cliff compressed document array profiles). This is an extended version of the bottom left panel of Figure 2.

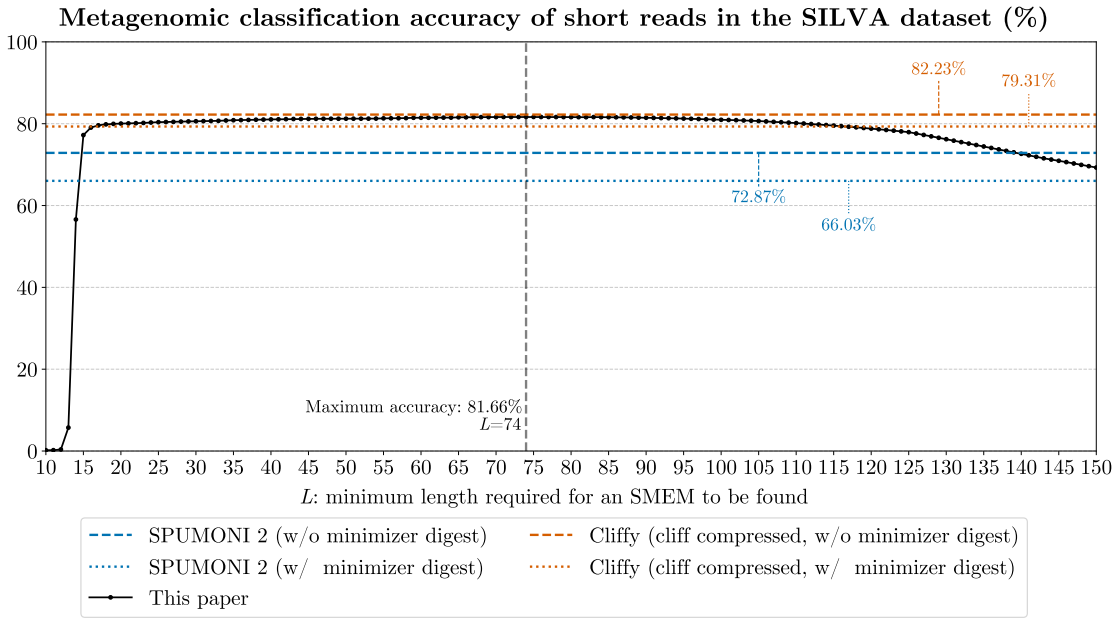

Figure S2: Analysis of metagenomic classification accuracy for matching 9 029 669 short reads to an index of 402 378 16S rRNA gene sequences spanning 9 118 metagenomic classes. We compare our approach for varying values of L, to SPUMONI 2 (without and with minimizer digests) and Cliffy with cliff compressed document array profiles (without and with minimizer digests). This is an extended version of the bottom left panel of Figure 3.

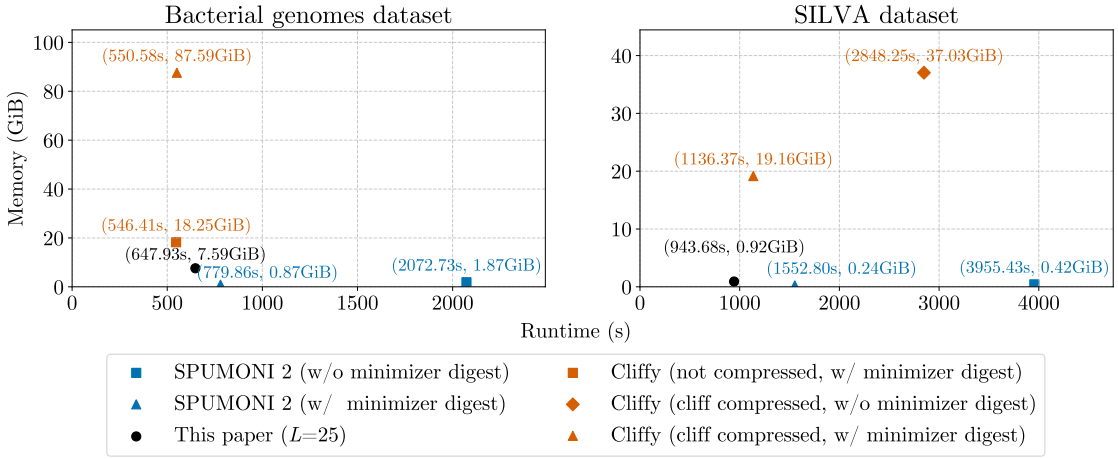

Figure S3: Memory versus runtime comparison of our approach ($L = 25$) with SPUMONI 2 and Cliffy, using all configurations the dataset allows. Shown are results for matching 4 000 000 short reads to an index of 8 165 bacterial genomes across 8 metagenomic classes (left); and for matching 9 029 669 short reads to an index of 402 378 16S rRNA gene sequences spanning 9 118 metagenomic classes (right).