Hong *et al. Algorithms for Molecular Biology* (2024) 19:15
https://doi.org/10.1186/s13015-024-00260-8

Algorithms for Molecular Biology

RESEARCH

Open Access

Pfp-fm: an accelerated FM-index

Aaron Hong[1], Marco Oliva[1], Dominik Köppl[2,3], Hideo Bannai[3], Christina Boucher[1*] and Travis Gagie[4]

Abstract

FM-indexes are crucial data structures in DNA alignment, but searching with them usually takes at least one random access per character in the query pattern. Ferragina and Fischer [1] observed in 2007 that word-based indexes often use fewer random accesses than character-based indexes, and thus support faster searches. Since DNA lacks natural word-boundaries, however, it is necessary to parse it somehow before applying word-based FM-indexing. In 2022, Deng et al. [2] proposed parsing genomic data by induced suffix sorting, and showed that the resulting word-based FM-indexes support faster counting queries than standard FM-indexes when patterns are a few thousand characters or longer. In this paper we show that using prefix-free parsing—which takes parameters that let us tune the average length of the phrases—instead of induced suffix sorting, gives a significant speedup for patterns of only a few hundred characters. We implement our method and demonstrate it is between 3 and 18 times faster than competing methods on queries to GRCh38, and is consistently faster on queries made to 25,000, 50,000 and 100,000 SARS-CoV-2 genomes. Hence, it seems our method accelerates the performance of count over all state-of-the-art methods with a moderate increase in the memory. The source code for `PFP-FM` is available at https://github.com/Aaron Hong1024/afm.

Keywords FM-index, Pangenomics, Word-based indexing, Random access

Introduction

The FM-index [3] is one of the most famous data structures in bioinformatics as it has been applied to countless applications in the analysis of biological data. Due to the long-term impact of this data structure, Burrows, Ferragina, and Manzini earned the 2022 ACM Paris Kanellakis Theory and Practice Award.[1] It is the data structure behind important read aligners—e.g., Bowtie [4] and BWA [5]—which take one or more reference genomes and build the FM-index for these genomes and use the resulting index to find short exact alignments between a set of reads and the reference(s) which then can be extended to approximate matches [4, 5]. Briefly, the FM-index consists of a sample of the suffix array (denoted as `SA`) and the Burrows–Wheeler transform (`BWT`) array. Given an input string S and a query pattern Q, `count` queries that answer the number of times the longest match of Q appears in S, can be efficiently supported using the `BWT`. To locate these occurrences a sampling of `SA` is used. Together the FM-index efficiently supports both `count` and `locate` queries. We mathematically define the `SA` and `BWT` in the next section.

There has been a plethora of research papers on reducing the size of the FM-index (see, e.g., [6–8]) and on speeding up queries. The basic query, `count`, returns the number of times a pattern Q appears in the indexed text S, but usually requires at least $|Q|$ random accesses to the `BWT` of S, which are usually much slower than the subsequent computations we perform on the information those accesses return. More specifically, a `count`

[1] https://awards.acm.org/kanellakis.

*Correspondence:
Christina Boucher
christinaboucher@ufl.edu
[1] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida 32611, USA
[2] Faculty of Engineering, University of Yamanashi, Kōfu 400-8510, Japan
[3] M &D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan
[4] Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada

query for Q uses rank queries at $|Q|$ positions in the BWT; if we answer these using a single wavelet tree for the whole BWT, then we may use a random access for every level we descend in the wavelet tree, or $\Omega(|Q| \log \sigma)$ random access in all, where σ is the size of the alphabet; if we break the BWT into blocks and use a separate wavelet tree for each block [7], we may need only one or a few random accesses per rank query, but the total number of random accesses is still likely to be $\Omega(|Q|)$. As far back as 2007, Ferragina and Fischer [1] addressed compressed indexes' reliance on random access and demonstrated that word-based indexes perform fewer random accesses than character-based indexes:*"The space reduction of the final word-based suffix array impacts also in their query time (i.e. less random access binary-search steps!), being faster by a factor of up to 3."*

Thus, one possibility of accelerating the random access to genomic data—where it is widely used—is to break up the sequences into words or phrases. In light of this insight, Deng, Hon, Köppl and Sadakane [2] in 2022 applied a grammar [9] that factorizes S into phrases based on the leftmost S-type suffixes (LMS) [10]. (Crescenzi et al. [11] proposed a similar idea earlier.) Unfortunately, one round of that LMS parsing leads to phrases that are generally too short, so they obtained a speedup only when Q was thousands of characters. The open problem was how to control the length of phrases with respect to the input to get longer phrases that would enable larger advances in the acceleration of the random access.

Here, we apply the concept of prefix-free parsing to the problem of accelerating count in the FM-index. Prefix-free parsing uses a rolling hash to first select a set of strings (referred to as *trigger strings*) that are used to define a parse of the input string S; i.e., the prefix-free parse is a parsing of S into phrases that begin and end at a trigger string and contain no other trigger string. All unique phrases are lexicographically sorted and stored in the dictionary of the prefix-free parse, which we denote as D. The prefix-free parse can be stored as an ordered list of the phrases' ranks in D. Hence, prefix-free parsing breaks up the input sequence into phrases, whose size is more controllable by the selection of the trigger strings. This leads to a more flexible acceleration than Deng et al. [2] obtained.

We assume that we have an input string S of length n. Now suppose we build an FM-index for S, an FM-index for the parse P, and a bitvector B of length n with 1's marking characters in the BWT of S that immediately precede phrase boundaries in S, i.e., that immediately precede a trigger string. We note that all the 1 s are bunched into at most as many runs as there are distinct trigger strings in S. Also, as long as the ranks of the phrases are in the same lexicographic order as the phrases themselves, we can use a rank query on the bitvector to map from the interval in the BWT of S for any pattern starting with a trigger string to the corresponding interval in the BWT of P, and vice versa with a select query. This means that, given a query pattern Q, we can backward search for Q character by character in the FM-index for S until we hit the left end of the rightmost trigger string in Q, then map into the BWT of P and backward search for Q phrase by phrase until we hit the left end of the leftmost trigger string in Q, then map back into the BWT of S and finish backward searching character by character again.

We implement this method, which we refer to as PFP-FM, and extensively compare against the FM-index implementation in sdsl [12], RLCSA [13], RLFM [6, 14], and FIGISS [2] using sets of SARS-CoV-2 genomes taken from the NCBI website, and the Genome Reference Consortium Human Build 38 with varying query string lengths. When we compare PFP-FM to FM-index in sdsl using 100,000 SARS-CoV-2 genomes, we witnessed that PFP-FM was able to perform between 2.1 times and 2.8 times more queries. In addition, PFP-FM was between 64.38% and 74.12%, 59.22% and 78.23%, and 49.10% and 90.70% faster than FIGISS, RLCSA, and RLFM, respectively, on 100,000 SARS-CoV-2 genomes. We evaluated the performance of PFP-FM on the Genome Reference Consortium Human Build 38, and witnessed that it was between 3.86 and 7.07, 2.92 and 18.07, and 10.14 and 25.46 times faster than RLCSA, RLFM, and FIGISS, respectively. With respect to construction time, PFP-FM had the most efficient construction time for all SARS-CoV-2 datasets and was the second fastest for Genome Reference Consortium Human Build 38. All methods used less than 60 GB for memory for construction on the SARS-CoV-2 datasets, making the construction feasible on any entry level commodity server—even the build for the 100,000 SARS-CoV-2 dataset. Construction for the Genome Reference Consortium Human Build 38 required between 26 GB and 71 GB for all methods, with our method using the most memory. In summary, we developed and implemented a method for accelerating the FM-index, and achieved an acceleration between 2 and 25 times, with the greatest acceleration witnessed with longer patterns. Thus, accelerated FM-index methods—such as the one developed in this paper—are highly applicable to finding very long matches (125 to 1000 in length) between query sequences and reference databases. As reads get longer and more accurate (i.e., Nanopore data), we will soon be prepared to align long reads to reference databases with efficiency that surpasses traditional FM-index based alignment methods. The source code is publicly available at https://github.com/AaronHong1024/afm.

Preliminaries

Basic definitions

A string S of length n is a finite sequence of symbols $S = S[0..n-1] = S[0] \cdots S[n-1]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$. We assume that the symbols can be unambiguously ordered. We denote by ε the empty string, and the length of S as $|S|$. Given a string S, we denote the reverse of S as $rev(S)$, i.e., $rev(S) = S[n-1] \cdots S[0]$.

We denote by $S[i..j]$ the substring $S[i] \cdots S[j]$ of S starting in position i and ending in position j, with $S[i..j] = \varepsilon$ if $i > j$. For a string S and $0 \leq i < n$, $S[0..i]$ is called the i-th prefix of S, and $S[i..n-1]$ is called the i-th suffix of S. We call a prefix $S[0..i]$ of S a *proper prefix* if $0 \leq i < n-1$. Similarly, we call a suffix $S[i..n-1]$ of S a *proper suffix* if $0 < i < n$.

Given a string S, a symbol $c \in \Sigma$, and an integer i, we define $S.\,\mathrm{rank}_c(i)$ (or simply `rank` if the context is clear) as the number of occurrences of c in $S[0..i-1]$. We also define $S.\,\mathrm{select}_c(i)$ as $\min(\{j-1 \mid S.\,\mathrm{rank}_c(j) = i\} \cup \{n\})$, i.e., the position in S of the i-th occurrence of c in S if it exists, and n otherwise. For a bitvector $B[0..n-1]$, that is a string over $\Sigma = \{0,1\}$, to ease the notation we will refer to $B.\,\mathrm{rank}_1(i)$ and $B.\,\mathrm{select}_1(i)$ as $B.\,\mathrm{rank}\,(i)$ and $B.\,\mathrm{select}\,(i)$, respectively.

SA, BWT, and backward search

We denote the *suffix array* [15] of a given a string $S[0..n-1]$ as SA_S, and define it to be the permutation of $\{0, \ldots, n-1\}$ such that $S[\,\mathrm{SA}_S[i]..n-1]$ is the i-th lexicographical smallest suffix of S. We refer to SA_S as SA when it is clear from the context. For technical reasons, we assume that the last symbol of the input string is $S[n-1] = \$$, which does not occur anywhere else in the string and is smaller than any other symbol.

We consider the matrix W containing all sorted rotations of S, called the `BWT` matrix of S, and let `F` and `L` be the first and the last column of the matrix. The last column defines the `BWT` array, i.e., $\mathrm{BWT} = \mathsf{L}$. Now let $\mathsf{C}[c]$ be the number of suffixes starting with a character smaller than c. We define the `LF`-mapping as $\mathrm{LF}\,(i,c) = \mathsf{C}[c] + \mathsf{BWT}.\,\mathrm{rank}_c(i)$ and $\mathrm{LF}\,(i) = \mathrm{LF}\,(i, \mathsf{BWT}[i])$. With the `LF`-mapping, it is possible to reconstruct the string S from its `BWT`. It is in fact sufficient to set an iterator $s = 0$ and $S[n-1] = \$$ and for each $i = n-2, \ldots, 0$ do $S[i] = \mathrm{BWT}\,[s]$ and $s = \mathrm{LF}\,(s)$. The `LF`-mapping can also be used to support `count` by performing the backward search, which we now describe.

Given a query pattern Q of length m, the *backward search* algorithm consists of m steps that preserve the following invariant: at the i-th step, p stores the position of the first row of W prefixed by $Q[i, m]$ while q stores the position of the last row of W prefixed by $Q[i, m]$. To advance from i to $i-1$, we use the `LF`-mapping on p and q, $p = \mathsf{C}[c] + \mathsf{BWT}.\,\mathrm{rank}_c(p)$ and $q = \mathsf{C}[c] + \mathrm{BWT}\,.\,\mathrm{rank}_c(q+1) - 1$.

FM-index and count queries

Given a query string $Q[0..m-1]$ and an input string $S[0..n-1]$, two fundamental queries are: (1) `count` which counts the number of occurrences of Q in S; (2) `locate` which finds the location of each of these matches in S. Ferragina and Manzini [3] showed that, by combining SA with the `BWT`, both `count` and `locate` can be efficiently supported. Briefly, backward search on the `BWT` is used to find the lexicographical range of the occurrences of Q in S; the size of this range is equal to `count`. The `SA` positions within this range are the positions where these occurrences are in S.

Prefix-free parsing

As we previously mentioned, the *Prefix-Free Parsing* (PFP) takes as input a string $S[0..n-1]$, and positive integers w and p, and produces a parse of S (denoted as P) and a dictionary (denoted as D) of all the unique substrings (or phrases) of the parse. Parameter w defines the length of the trigger strings and parameter p influences the rolling-hash function. We briefly go over the algorithm for producing this dictionary D and parse P. First, we assume there exists two symbols, say # and \$, which are not contained in Σ and are lexicographically smaller than any symbol in Σ. Next, we let T be an arbitrary set of w-length strings over Σ and call it the set of *trigger strings*. As mentioned before, we assume that $S[n-1] = \$$ and consider S to be cyclic, i.e., for all i, $S[i] = S[i \bmod n]$. Furthermore, we assume that $\$S[0..w-2] = S[n-1..n+w-2] \in T$, i.e., the substring of length w that begins with \$ is a trigger string.

We let the dictionary $\mathrm{D} = \{d_1, .., d_{|\mathrm{D}|}\}$ be a (lexicographically sorted) maximum set of substrings of S such that the following holds for each d_i: i) exactly one proper prefix of d_i is contained in T, ii) exactly one proper suffix of d_i is contained in T, iii) and no other substring of d_i is in T. These properties allow for the `SA` and `BWT` to be constructed since the lexicographical placement of each rotation of the input string can be identified unambiguously from `D` and `P` [16–18]. An important consequence of the definition is that D is prefix-free, i.e., for any $i \neq j$, d_i cannot be a prefix of d_j.

Since we assumed $S[n-1..n+w-2] \in T$, we can construct D by scanning $S' = \$S[0..n-2]S[n-1..n+w-2]$ to find all occurrences of T and adding to D each substring of S' that starts and ends at a trigger string being

$$\begin{bmatrix} 0 & 2 & 4 & 3 & 1 & 5 \\ 1 & 5 & 0 & 2 & 4 & 3 \\ 2 & 4 & 3 & 1 & 5 & 0 \\ 3 & 1 & 5 & 0 & 2 & 4 \\ 4 & 3 & 1 & 5 & 0 & 2 \\ 5 & 0 & 2 & 4 & 3 & 1 \end{bmatrix} \qquad \begin{array}{ll} 0: & \texttt{\$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT} \\ 1: & \texttt{AAGACATATGAT\$TCCAGAAGAGTATCTCCTCGACATGTTG} \\ 2: & \texttt{AAGAGTATCTCCTCGACATGTTGAAGACATATGAT\$TCCAG} \\ 3: & \texttt{CGACATGTTGAAGACATATGAT\$TCCAGAAGAGTATCTCCT} \\ 4: & \texttt{TATCTCCTCGACATGTTGAAGACATATGAT\$TCCAGAAGAG} \\ 5: & \texttt{TATGAT\$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA} \end{array}$$

Fig. 1 The BWT matrix for our prefix-free parse P (left) and the cyclic shifts of S that start with a trigger string (right), in lexicographic order

inclusive of the starting and ending trigger string. We can also construct the list of occurrences of D in S', which defines the parse P.

We choose T by a Karp-Rabin fingerprint f of strings of length w. We slide a window of length w over S', and for each length w substring r of S', include r in T if and only if $f(r) \equiv 0 \pmod{p}$ or $r = S[n-1..n+w-2]$. Let $0 = s_0 < \cdots < s_{k-1}$ be the positions in S' such that for any $0 \le i < k$, $S'[s_i..s_i + w - 1] \in T$. The dictionary is $\mathrm{D} = \{S'[s_i..s_{i+1} + w - 1] \mid i = 0, \ldots, k-1\}$, and the parse is defined to be the sequence of lexicographic ranks in D of the substrings $S'[s_0..s_1 + w - 1], \ldots, S'[s_{k-2}..s_{k-1} + w - 1]$.

As an example, suppose we have $S' =$ \$AGACGACT#AGATACT#AGATTCGAGACGAC\$A, where the trigger strings are highlighted in red, blue, or green. It follows that we have D = {\$AGAC, AC\$A, ACGAC, ACT#AGA TAC, ACT#AGA TTC, TCG AGA C} and P = 0, 2, 3, 4, 5, 2, 1. In our experiment, as illustrated in Fig. 3, we observed that increasing the value of w usually decreases the average phrase length. Conversely, increasing p usually increases the average phrase length. We note, however, that these trends may vary in real-world applications.

Methods

We now make use of the prefix-free parsing reviewed in section "Prefix-free parsing" to build a word-based FM-index in a manner in which the lengths of the phrases can be controlled via the parameters w and p. To explain our data structure, we first describe the components of our data structure, and then explain how to support count queries in a manner that is more efficient than the standard FM-index.

Data structure design

It is easiest to explain our two-level design with an example, so consider a text

$$S[0..n-1] = \texttt{TCCAGAAGAGTATCT}\\ \texttt{CCTCGACATGTTGA}\\ \texttt{AGACATATGAT\$}$$

of length $n = 41$ that is terminated by a special end-of-string character \$ lexicographically less than the rest of the alphabet. Suppose we parse S using $w = 2$ and a Karp-Rabin hash function such that the trigger strings occurring in S are AA, CG and TA. We consider S as cyclic, and we have $\$S[0..w-2] =$ \$ T as a special trigger string, so the dictionary D is

$$\begin{aligned} \mathrm{D}\,[0..5] = \{&\texttt{\$TCC AGA A}, \texttt{AAG ACA TA},\\ &\texttt{AAG AGT A}, \texttt{CGA CAT GTT GAA},\\ &\texttt{TAT CTC CTCG}, \texttt{TAT GAT \$T}\}, \end{aligned}$$

with the phrases sorted in lexicographic order. (Recall that phrases consecutive in S overlap by $w = 2$ characters.) If we start parsing at the \$, then the prefix-free parse for S is

$$\mathrm{P}\,[0..5] = (0, 2, 4, 3, 1, 5),$$

where each element (or phrase ID) in P is the lexicographic rank of the phrase in D.

Next, we consider the BWT matrix for P. Figure 1 illustrates the BWT matrix of P for our example. We note that since there is only one \$ in S, it follows that there is only one 0 in P; we can regard this 0 as the end-of-string character for (a suitable rotation of) P corresponding to \$ in S. If we take the i-th row of this matrix and replace the phrase IDs by the phrases themselves, collapsing overlaps, then we get the lexicographically i-th cyclic shift of S that start with a trigger string, as shown on the right of the figure. This is one of the key insights that we will use later on.

Lemma 1 *The lexicographic order of rotations of* P *correspond to the lexicographic order of their corresponding rotations of S.*

Proof The characters of P are the phrase IDs that act as meta-characters. Since the meta-characters inherit the lexicographic rank of their underlying characters, and due to the prefix-freeness of the phrases, the suffix array of P permutes the meta-characters of P in the same way as the suffix array of S permutes the phrases of S. This

i	SA[i]	$B[i]$	T[SA[i]..(SA[i] − 1) mod n]	BWT[i]
0	40	1	$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT	T
1	28	1	AAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTG	G
2	5	1	AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG	G
3	31	0	ACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAG	G
4	20	0	ACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCG	G
5	3	0	AGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCC	C
6	29	0	AGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGA	A
7	6	0	AGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGA	A
8	8	0	AGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAG	G
9	38	0	AT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATG	G
10	33	0	ATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGAC	C
11	11	0	ATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGT	T
12	35	0	ATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACAT	T
13	22	0	ATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGAC	C
14	2	0	CAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TC	C
15	32	0	CATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGA	A
16	21	0	CATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGA	A
17	1	0	CCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$T	T
18	15	0	CCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCT	T
19	18	1	CGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCT	T
20	13	0	CTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTAT	T
21	16	0	CTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTC	C
22	27	0	GAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTT	T
23	4	0	GAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCA	A
24	30	0	GACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAA	A
25	19	0	GACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTC	C
26	7	0	GAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAA	A
27	37	0	GAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATAT	T
28	9	0	GTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGA	A
29	24	0	GTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACAT	T
30	39	0	T$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGA	A
31	10	1	TATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAG	G
32	34	1	TATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA	A
33	0	0	TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$	$
34	14	0	TCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATC	C
35	17	0	TCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCC	C
36	12	0	TCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTA	A
37	26	0	TGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGT	T
38	36	0	TGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATA	A
39	23	0	TGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACA	A
40	25	0	TTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATG	G

Fig. 2 The SA, BWT matrix and BWT of T, together with the bitvector B in which 1 s indicate rows of the matrix starting with trigger strings. The BWT is highlighted in red, while the columns marked by 1 s are highlighted in blue

means that the order of the phrases in the BWT of S is the same as the order of the phrase IDs in P. □

Next, we let $B[0..n-1]$ be a bitvector marking these cyclic shifts' lexicographic rank among all cyclic shifts of S, i.e., where they are among the rows of the `BWT` matrix of S. Figure 2 shows the `SA`, `BWT` matrix and `BWT` of S, together with B; we highlight the `BWT`—the last column of the matrix—in red, and the cyclic shifts from Fig. 1 are highlighted in blue. We note that B contains at most one run of 1's for each distinct trigger string in S, so it is usually highly run-length compressible in practice, While this compressibility is partly due to the generally small size of D observed in repetitive data, it is also influenced by the limited number of distinct trigger strings. For example, for 200 copies of of chromosome 19 the size of D was 0.16 GB and with 2200 additional copies the size of D was still less than 1 GB, i.e., 0.56 GB [19]. This suggests a compact representation, especially in repetitive sequences where distinct strings of length w are fewer, as approximately a $1/p$ fraction of these strings will be trigger strings.

In addition to the bitvector, we store a hash function h on phrases and a map M from the hashes of the phrases in D to those phrases' lexicographic ranks, which are their phrase IDs; M returns NULL when given any other key, where NULL semantically represents an invalid ID@. Therefore, in total, we build the FM-index for S, the FM-index for P, the bitvector B marking the cyclic rotations, the hash function h on the phrases and the map M. For our example, suppose

$$\begin{aligned} h(\texttt{\$TCCAGAA}) &= 91785 \quad M(91785) = 0 \\ h(\texttt{AAGACATA}) &= 34865 \quad M(34865) = 1 \\ h(\texttt{AAGAGTA}) &= 49428 \quad M(49428) = 2 \\ h(\texttt{CGACATGTTGAA}) &= 98759 \quad M(98759) = 3 \\ h(\texttt{TATCTCCTCG}) &= 37298 \quad M(37298) = 4 \\ h(\texttt{TATGAT\$T}) &= 68764 \quad M(68764) = 5 \end{aligned}$$

and $M(x) = \text{NULL}$ for any other value of x.

If we choose the range of h to be reasonably large then we can still store M in space proportional to the number of phrases in D with a reasonably constant coefficient and evaluate $M(h(\cdot))$ in constant time with high probability, but the probability is negligible that $M(h(\gamma)) \neq \text{NULL}$ for any particular string γ not in D. This means that in practice we can use $M(h(\cdot))$ as a membership dictionary for D, and not store D itself.

Query support

Next, given the data structure that we define above, we describe how to support `count` queries for a given pattern Q. We begin by parsing Q using the same Karp-Rabin hash we used to parse S, implying that we will have all the same trigger strings as we did before and possibly additional ones that did not occur in S. However, we will not consider Q to be cyclic nor assume an end-of-string symbol that would assure that Q starts and ends with a trigger string.

If Q is a substring of S, then, since Q contains the same trigger strings as its corresponding occurrence in S, the sequence of phrases induced by the trigger strings in Q must be a substring of the sequence of phrases of S. Together with the prefix and suffix of Q that are a suffix and prefix of the phrases in S to the left and right of the shared phrases, we call this the partial encoding of Q, defined formally as follows.

Definition 1 (partial encoding) Given a substring $S[i..j]$ of S, the *partial encoding* of $S[i..j]$ is defined as follows: If no trigger string occurs in $S[i..j]$, then the partial encoding of $S[i..j]$ is simply $S[i..j]$ itself. Otherwise, the partial

encoding of $S[i..j]$ is the concatenation of: (1) the shortest prefix α of $S[i..j]$ that does not start with a trigger string and ends with a trigger string, followed by (2) the sequence of phrase IDs of phrases completely contained in $S[i..j]$, followed by (3) the shortest suffix β of $S[i..j]$ that begins with a trigger string and does not end with a trigger string.

So the partial encoding partitions $S[i..j]$ into a prefix α, a list of phrase IDs, and a suffix β. If $S[i..j]$ begins (respectively ends) with a trigger string, then α (respectively β) is the empty string.

Parsing Q can be done in time linear in the length of Q.

Lemma 2 *We can represent M with a data structure taking space (in words) proportional to the number of distinct phrases in* D. *Given a query pattern Q, this data structure returns* NULL *with high probability if Q contains a complete phrase that does not occur in S. Otherwise (complete phrases of Q occur in S), it returns the partial encoding of Q. In either case, this query takes $O(|Q|)$ time.*

Proof We keep the Karp-Rabin (KR) hashes of the phrases in D, with the range of the KR hash function mapping to $[1..n^3]$ so the hashes each fit in $O(\log n)$ bits. We also keep a constant-time map (implemented as a hash table with a hash function that is perfect for the phrases in D) from the KR hashes of the phrases in D to their IDs, that returns NULL given any value that is not a KR hash of a phrase in D. We set M to be the map composed with the KR hash function.

Given Q, we scan it to find the trigger strings in it, and convert it into a sequence of substrings consisting of: (a) the prefix α of Q ending at the right end of the first trigger string in Q; (b) a sequence of PFP phrases, each starting and ending with a trigger string with no trigger string in between; and (c) the suffix β of Q starting at the left end of the last trigger string in Q.

We apply M to every complete phrase in (b). If M returns NULL for any complete phrase in (b), then that phrase does not occur in S, so we return NULL; otherwise, we return α, the sequence of phrase IDs M returned for the phrases in (b), and β.

Notice that, if a phrase in Q is in S, then M will map it to its lexicographic rank in D; otherwise, the probability the KR hash of any particular phrase in Q but not in D collides with the KR hash of a phrase in D, is at most $n/n^3 = 1/n^2$. It follows that, if Q contains a complete phrase that does not occur in S, then we return NULL with high probability; otherwise, we return Q's partial encoding. □

Corollary 1 *If we allow $O(|Q|)$ query time with high probability, then we can modify M to always report* NULL *when Q contains a complete phrase not in S.*

Proof We augment each Karp-Rabin (KR) hash stored in the hash table with the actual characters of its phrase such that we can check, character by character, whether a matched phrase of Q is indeed in D. In case of a collision we recompute the KR hashes of D and rebuild the hash table. That is possible since we are free to choose different Karp-Rabin fingerprints for the phrases in D. □

Continuing from our example above where the trigger strings are AA, CG and TA, suppose we have a query pattern Q,

$$Q[0..34] = \texttt{CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT}$$

we can compute the parse of Q to obtain the following parse string

$$\texttt{CAGAA}, \texttt{AAGAGTA}, \texttt{TATCTCCTCG},$$
$$\texttt{CGACATGTTGAA}, \texttt{AAGACATA}, \texttt{TAT}.$$

Next, we use $M(h(\cdot))$ to map the complete phrases of this parse of Q to their phrase IDs—which are their rank values in D. If any complete phrase maps to NULL then we know Q does not occur in T. Using our example, we have the partial encoding

$$\texttt{CAGAA}, 2, 4, 3, 1, \texttt{TAT}.$$

Next, we consider all possible cases matching Q with an occurrence in S. First, we consider the case that the last substring β in our parse of Q ends with a trigger string, which implies that it is a complete phrase. Here, we can immediately start backward searching for the parse of Q in the FM-index for P. Next, if β is not a complete phrase then we backward search for β in the FM-index for S. If this backward search for β returns nothing then we know Q does not occur in S. If the backward search for β returns an interval in the BWT of P that is not contained in the BWT interval for a trigger string then β does not start with a trigger string so $Q = \beta$ and we are done backward searching for Q.

Finally, we consider the case when β is a proper prefix of a phrase and the backward search for β returns a BWT_S interval contained in the BWT_S interval for a trigger string. In our example, $\beta = \texttt{TAT}$ and our backward search for β in the FM-index for S returns the interval $\mathrm{BWT}_S[31..32]$, which is the interval for the trigger string TA. Next, we use B to map the interval for β in the BWT_S to the interval in the $\mathrm{BWT}_{\mathrm{P}}$ that corresponds to the cyclic shifts of S starting with β.

Lemma 3 *We can store, in space (in words) proportional to the number of distinct trigger strings in S, a data structure B with which,*

- Given the lexicographic range of suffixes of S starting with a string β such that β starts with a trigger string and contains no other trigger string, in $O(\log\log n)$ time we can find the lexicographic range of suffixes of P starting with phrases that start with β;
- Given a lexicographic range of suffixes of P such that the corresponding suffixes of S all start with the same trigger string, in $O(\log\log n)$ time we can find the lexicographic range of those corresponding suffixes of S.

Proof Let $B[0..n-1]$ be a bitvector with 1 s marking the lexicographic ranks of suffixes of S starting with trigger strings. There are at most as many runs of 1 s in B as there are distinct trigger strings in S, so we can store B in space proportional to that number and support rank and select operations on it in $O(\log\log n)$ time (e.g. with the data structure of [20]).
If $\mathrm{BWT}_S[i..j]$ contains the characters immediately preceding, in S, occurrences of a string β that starts with a trigger string and contains no other trigger strings, then $\mathrm{BWT}_{\mathrm{P}}[B.\,\mathrm{rank}_1(i)..B.\,\mathrm{rank}_1(j)]$ contains the phrase IDs immediately preceding, in P, the IDs of phrases starting with β.

If $\mathrm{BWT}_{\mathrm{P}}[i..j]$ contains the phrase IDs immediately preceding, in P, suffixes of P such that the corresponding suffixes of S all start with the same trigger string, then $\mathrm{BWT}_S[B.\,\mathrm{select}_1(i+1)..B.\,\mathrm{select}_1(j+1)]$ contains the characters immediately preceding the corresponding suffixes of S.

The correctness follows from Lemma 1. □

Continuing with our example, mapping $\mathrm{BWT}_S[31..32]$ to $\mathrm{BWT}_{\mathrm{P}}$ yields the following interval:

$$\mathrm{BWT}_{\mathrm{P}}[B.\,\mathrm{rank}_1(31), B.\,\mathrm{rank}_1(32)] = \mathrm{BWT}_{\mathrm{P}}[4..5]$$

as shown in Fig. 1. Starting from this interval in $\mathrm{BWT}_{\mathrm{P}}$, we now backward search in the FM-index for P for the sequence of complete phrase IDs in the parse of Q. In our example, we have the interval $\mathrm{BWT}_{\mathrm{P}}[4..5]$ which yields the following phrase IDs: 2 4 3 1.

If this backward search in the FM-index for P returns nothing, then we know Q does not occur in S. Otherwise, it returns the interval in $\mathrm{BWT}_{\mathrm{P}}$ corresponding to cyclic shifts of S starting with the suffix of Q that starts with Q's first complete phrase. In our example, if we start with $\mathrm{BWT}_{\mathrm{P}}[4..5]$ and backward search for 2 4 3 1 then we obtain $\mathrm{BWT}_{\mathrm{P}}[2]$, which corresponds to the cyclic shift

```
AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG
```

of S that starts with the suffix

```
AAGAGTATCTCCTCGACATGTTGAAGACATAT
```

of Q that is parsed into 2, 4, 3, 1, `TAT`.

To finish our search for Q, we use B to map the interval in $\mathrm{BWT}_{\mathrm{P}}$ to the corresponding interval in the BWT_S, which is the interval of rows in the `BWT` matrix for S which start with the suffix of Q we have sought so far. In our example, we have that $\mathrm{BWT}_{\mathrm{P}}[2]$ maps to

$$\mathrm{BWT}_S[B.\,\mathrm{select}_1(2+1)] = \mathrm{BWT}_S[2].$$

We note that our examples contain `BWT` intervals with only one entry because our example is so small, but in general they are longer. If the first substring α in our parse of Q is a complete phrase then we are done backward searching for Q. Otherwise, we start with this interval in BWT_S and backward search for α in the FM-index for S, except that we ignore the last w characters of α (which we have already sought, as they are also contained in the next phrase in the parse of Q).

In our example, $\alpha =$ `CAGAA` so, starting with $\mathrm{BWT}_S[2]$ we backward search for `CAG`, which returns the interval $\mathrm{BWT}_S[14]$. As shown in Fig. 2,

$$S[\,\mathrm{SA}\,[4]..n] = S[2..n] = \texttt{CAGAAGAGTATCTCCTCGACATGT}\ \texttt{TGAAGACATATGAT\$}$$

indeed starts with

$$Q = \texttt{Q = CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT.}$$

This concludes our explanation of `count`.

To conclude, we give some intuition as to why we expect that our two-level FM-index is faster in practice than standard backward search. Following the reasoning of Deng et al. [2], on the one hand, standard backward search takes linear time in the length of Q and usually uses at least one random access to the BWT of S per character in Q. On the other hand, prefix-free parsing Q, like the LMS-parsing of Deng et al., takes linear time but does not use random access to S or the BWT of S; backward search in the FM-index of S is the same as standard backward search but we use it only for the first and last substrings in the parse of Q. Backward search in the FM-index for P is likely to use about lg | D | random accesses

for each complete phrase in the parse of Q: the BWT of P is over an effective alphabet whose size is the number of phrases in D. Therefore, a balanced wavelet tree to support rank on that BWT should have depth about $\lg |D|$ and we should use at most about one random access for each level in the tree.

In summary, if we can find settings of the prefix-free parsing parameters w and p such that

- Most query patterns will span several phrases,
- Most phrases in those patterns are fairly long,
- $\lg |D|$ is significantly smaller than those phrases' average length, then the extra cost of parsing Q should be more than offset by using fewer random accesses.

Results

We implemented our algorithm and measured its performance against all known competing methods. We ran all experiments on a server with AMD EPYC 75F3 CPU with Red Hat Enterprise Linux 7.7 (64 bit, kernel 3.10.0). The compiler was g++ version 12.2.0. The running time and memory usage was recorded by Snakemake benchmark facility [21]. We set a memory limitation of 128 GB and a time limitation of 24 h.

Datasets We used the following datasets. First, we considered sets of SARS-CoV-2 genomes taken from the NCBI website. We used three collections of 25, 000, 50, 000, and 100, 000 SARS-CoV-2 genomes from EMBL-EBI's COVID-19 data portal. Each collection is a superset of the previous. We denote these as SARS-25k, and SARS-50k, SARS-100k. Next, we considered a single human reference genome, which we denote as GRCh38, downloaded from NCBI. We report the size of the datasets as the number of characters in each in Table 1. We denote n as the number of characters.

Implementation We implemented our method in C++ 11 using the sdsl-lite library [12] and extended the prefix-free parsing method of Oliva, whose source code is publicly available here https://github.com/marco-oliva/pfp. The source code for PFP-FM is available at https://github.com/AaronHong1024/afm.

Competing methods We compared PFP-FM against the following methods the standard FM-index found in sdsl-lite library [12], RLCSA [13], RLFM [6, 14], Bowtie [4], Bowtie2 [4, 22, 23] and FIGISS [2]. The source code of RLCSA and FIGISS is publicly available, while RLFM is provided only as an executable. We performed the comparison by selecting 1000 strings at random of the specified length from the FASTA file containing all the genomes, performing the count operation on each query pattern, and measuring the time usage for all the methods under consideration.

As a side note, FIGISS and RLCSA only support count queries where the string is provided in an input text file. More specifically, the original FIGISS implementation supports counting with the entire content of a file treated as a single pattern. To overcome this limitation, we modified the source code to enable the processing of multiple query patterns within a single file. In addition to the time required for answering count, we measured the time and memory required to construct the data structure.

Acceleration versus baseline

In this subsection, we compare PFP-FM versus the standard FM-index in sdsl with varying values of window size (w) and modulo value (p), and varying the length of the query pattern. We calculated the number of count queries performed per CPU second with PFP-FM versus the standard FM-index. We generated heatmaps that illustrate the number of count queries of PFP-FM versus sdsl for various lengths of query patterns, namely, 125, 250, 500, and 1000. We performed this for both SARS-CoV-2 set of genomes and GRCh38 human reference genome. Figure 3 shows the resulting heatmaps for SARS-100K. As shown in this figure, PFP-FM was between 2.178 and 2.845 times faster than the standard FM-index with the optimal values of w and p. In particular, an optimal performance gain of 2.6, 2.3, 2.2, and 2.9 was witnessed for pattern lengths of 125, 250, 500, and 1,000, respectively. The (w, p) pairs that correspond to these results are (6, 50), (6, 30), (8, 50), and (8, 50). Additionally, as depicted in Fig. 4, which focuses on the GRCh38 dataset, the speed of PFP-FM ranges from 1.672 to 2.943 times faster than that of the standard FM-index when optimal values of w and p are used. For pattern lengths of 125, 250, 500, and 1,000, the acceleration factors achieved by PFP-FM are 1.96, 1.81, 2.45, and 2.94, corresponding to these lengths. The specific (w, p) pairs for these improvements are (4, 50) for the 125 pattern length, (8, 50) for 250, (4, 50) for 500, and (6, 40) for the 1000 length. As detailed in Table 1, these outcomes were obtained under conditions of comparable memory usage and constructing time.

Results on SARS-CoV-2 genomes

We used the optimal parameters that were obtained from the previous experiment for this section. We constructed the index using these parameters for each SARS-CoV-2 dataset and assessed the time consumption for performing 1000 count queries using all competing methods and PFP-FM. We illustrate the result of this experiment in Fig. 5, where PFP-FM consistently exhibits the lowest time consumption. For the SARS-25K dataset, the time consumption of FIGISS

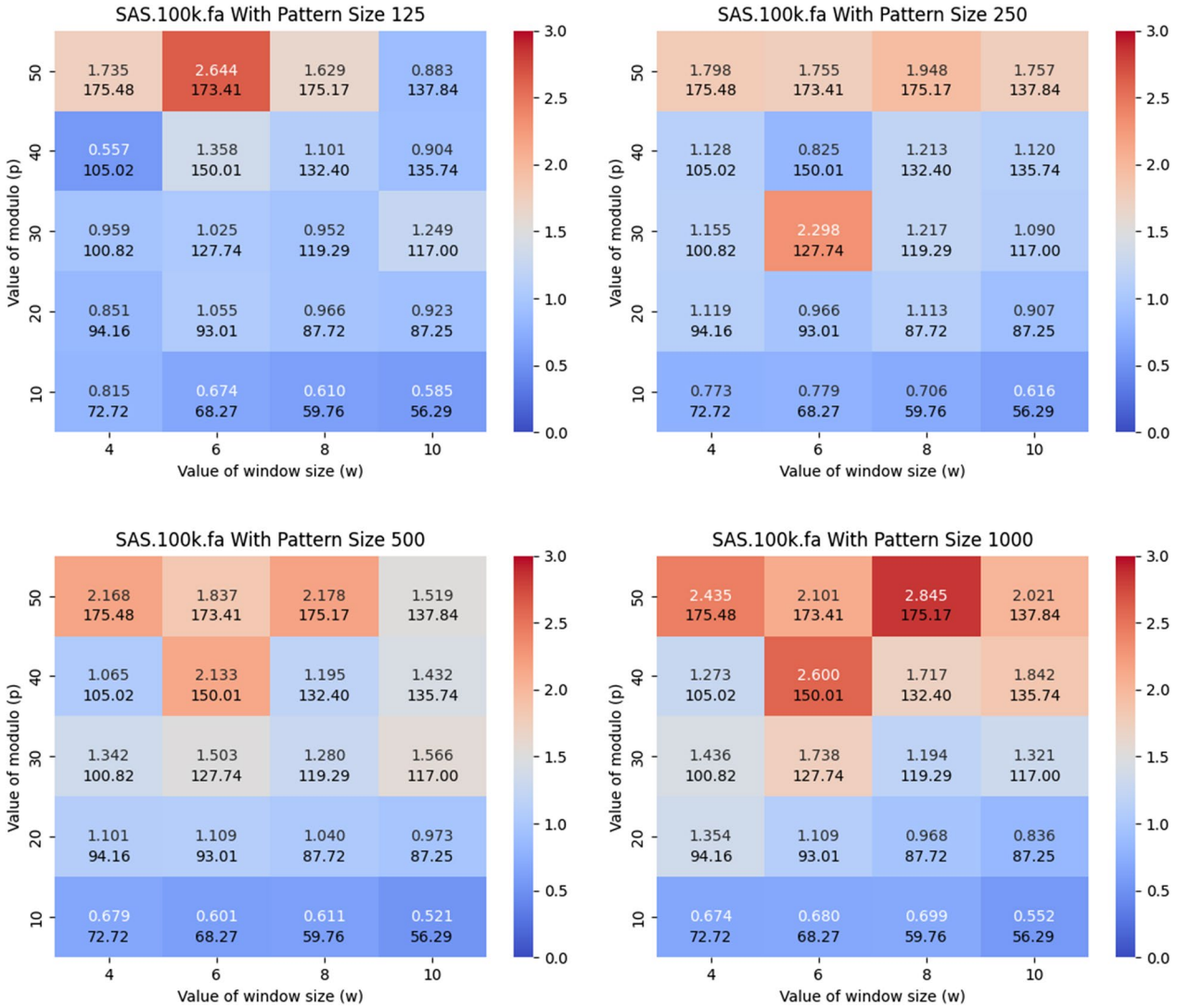

Fig. 3 Illustration of the impact of *w*, *p* and the length of the query pattern on the acceleration of the FM-index. Here, we used the `SARS-100K` dataset and varied the length of the query pattern to be equal to 125, 250, 500, and 1000. The y-axis corresponds to *p* and the x-axis corresponds to *w*. The heatmap illustrates the number of queries that can be performed in a CPU second with the acceleration versus the standard FM-index from `sdsl`, which employs a Huffman-shaped wavelet tree, i.e., `PFP-FM / sdsl`. The second value in each block represents the average length of the phrases

was between 451% and 568% higher than our method. And the time consumption of `RLCSA` and `RLFM` was between 780% and 1598%, and 842% and 1705% more than `PFP-FM`, respectively. The performance of `FIGISS` surpasses that of `RLFM` and `RLCSA` when using the SARS-25k dataset; however for the larger datasets `FIGISS` and `RLCSA` converge in their performance. Neither method was substantially better than the other. In addition, on the larger datasets, when the query pattern length was 125 and 250, `RLFM` performed better than `RLCSA` and `FIGISS` but was slower for the other query lengths. Hence, it is very clear that `PFP-FM` accelerates the performance of `count` over all state-of-the-art methods.

The gap in performance between `PFP-FM` and the competing methods increased with the dataset size. For `SARS-50K`, `FIGISS`, `RLCSA` and `RLFM` were between 3.65 and 13.44, 3.65 and 16.08, and 4.25 and 12.39 times slower, respectively. For `SARS-100K`, `FIGISS`, `RLCSA` and `RLFM` were between 2.81 and 3.86, 2.45 and 4.59, and 1.96 and 10.75 times slower, respectively.

Next, we consider the time and memory required for construction, which is given in Table 1. Our experiments revealed that all methods used less than 60 GB

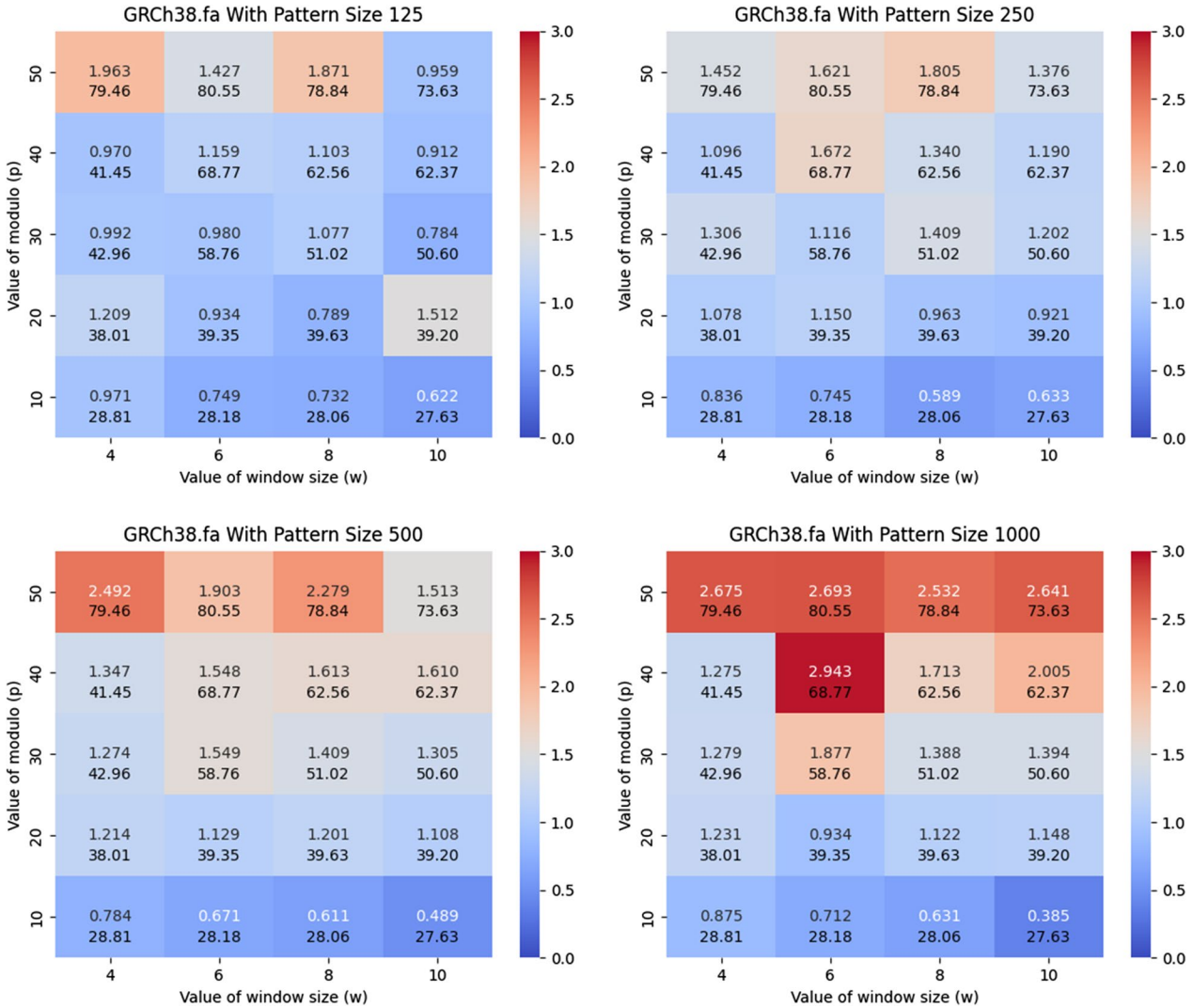

Fig. 4 Illustration of the impact of *w*, *p* and the length of the query pattern on the acceleration of the FM-index. Here, we used the `GRCh38` dataset and varied the length of the query pattern to be equal to 125, 250, 500, and 1000. The y-axis corresponds to *p* and the x-axis corresponds to *w*. The heatmap illustrates the number of queries that can be performed in a CPU second with the acceleration versus the standard FM-index from `sdsl`, which employs a Huffman-shaped wavelet tree, i.e., `PFP-FM` / `sdsl`. The second value in each block represents the average length of the phrases

of memory on all SARS-CoV-2 datasets; `PFP-FM` used the most memory with the peak being 54 GB on the `SARS-100K` dataset. Yet, `PFP-FM` exhibited the most efficient construction time across all datasets for generating the FM-index, and this gap in the time grew with the size of the dataset. More specifically, for the `SARS-100K` dataset, `PFP-FM` used 71.04%, 65.81%, and 73.41% less time compared to other methods. In summary, `PFP-FM` significantly accelerated the `count` time, and had the fastest construction time. All methods used less than 60 GB, which is available on most commodity servers. In comparison with Bowtie [4] and Bowtie 2 [22, 24], our study finds a notable trade-off for highly repetitive datasets; while Bowtie and Bowtie2 are more memory-efficient, it significantly increases processing time. In our experiments, Bowtie required at least ten times more time in constructing the index than our approach. We note that these methods have significant larger capability than our methods so this comparison should approached codicillary.

We observe in these experiments that the `PFP-FM` algorithm manifests a moderately larger index size compared to other algorithms. This can be attributed to the `PFP-FM` algorithm's methodology of storing the

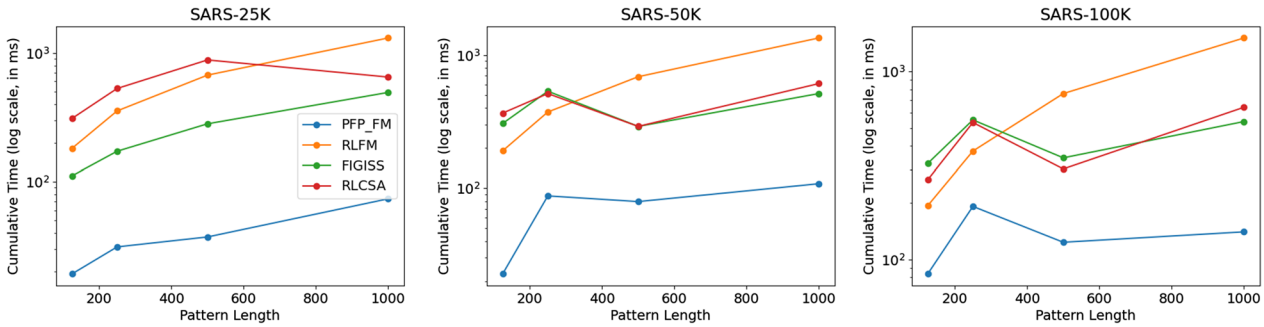

Fig. 5 Illustration of the impact of the dataset size, and the length of the query pattern on the query time for answering `count`. We vary the length of the query pattern to be equal to 125, 250, 500, and 1000, and report the times for `SARS-25K`, `SARS-50K`, and `SARS-100K`. We illustrate the cumulative time required to perform 1,000 `count` queries. The y-axis is in log scale

Fm-index for both string `S` and parse P, the suffix array (SA) for `S`, and a bitvector *B*. Of these, the SA contributes the most to the overall size. In an effort to mitigate this, the SA was substituted with a Compressed Suffix Array (`CSA`), as delineated in the work of Grossi et al. (2014) [12]. This substitution significantly diminishes the index size. Consequently, this modified approach has been designated as `PFP-FM-CSA`. As indicated in Table 1, when applied to the SARS-CoV-2 dataset, the `PFP-FM-CSA` algorithm requires more memory and time for construction. However, it notably reduces the index size to one-third of what is observed with the original `PFP-FM` algorithm.

We next assessed the query times of the `PFP-FM-CSA` algorithm in comparison with other algorithms, which is shown in Fig. 6. For the `GRCH38` dataset, due to the computation time of `PFP-FM-CSA` exceeding two days, we did not record the data. In other datasets, the `PFP-FM-CSA` showed superior performance, outpacing all other algorithms. Specifically, relative to the `PFP-FM` algorithm, `PFP-FM-CSA` was faster by 39.31% and 29.28%.

Thus, in conclusion, we see a trade-off between memory usage and construction time but note that this work contributes to the growing interested indexing data structures in bioinformatics. As novel implementations of construction algorithms for compressed

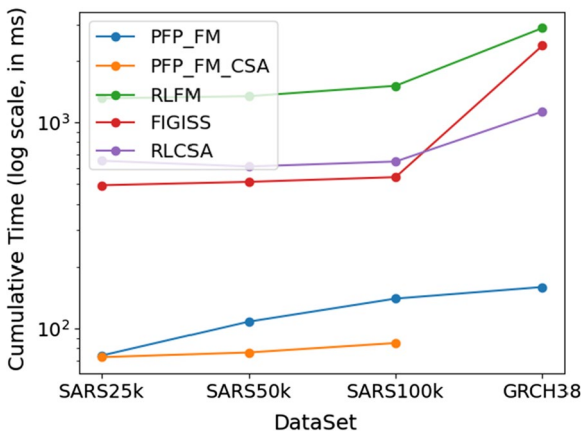

Fig. 6 Impact of Dataset Size and Query Pattern Length on Query Execution Time. This figure presents a comparative analysis of query times for `count` operations across various datasets: `SARS-25K`, `SARS-50K`, `SARS-100K`, and `GRCH38`, using a consistent query pattern length of 1,000. The cumulative time required for executing 1000 `count` queries is illustrated, with the y-axis representing time in log scale. Note that for the `GRCH38` dataset, due to the computation time exceeding two days, the data were not recorded

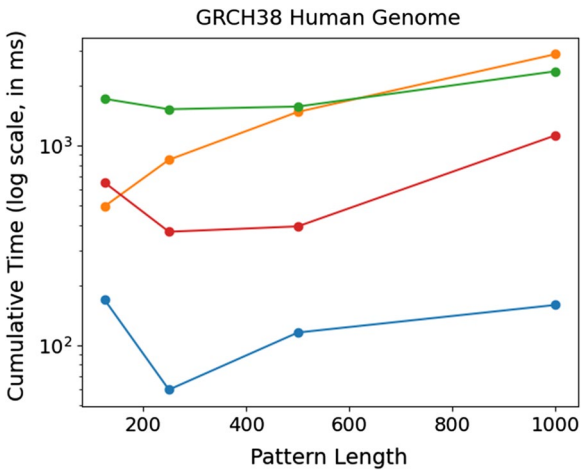

Fig. 7 Comparison of query times for `count` between the described solutions when varying the length of the query pattern. For each pattern length equal to 125, 250, 500, and 1000, we report the times for the `GRCH38` dataset. We plot the cumulative time required to perform 1000 `count` queries. The y-axis is in log scale. `PFP-FM` is shown in blue, `RLFM` is shown in orange, `RLCSA` is shown in red, and `FIGISS` is shown in green

Table 1 Comparison of the construction performance with the construction time and memory for all datasets

Dataset	*n*	Method	CONSTRUCT MEMORY	INDEX SIZE	CONSTRUCT TIME
SARS-25k	751,526,774	RLCSA	9.90	0.026	322.85
		RLFM	3.47	0.136	363.74
		FIGISS	4.89	0.003	378.49
		PFP-FM	12.99	4.318	117.29
		PFP-FM-CSA	15.68	1.689	772.98
		FM-index	13.35	4.399	120.08
		Bowtie	3.55	0.47	7851.35
		Bowtie2	3.54	0.59	6847.03
SARS-50k	1,503,252,577	RLCSA	19.88	0.051	679.89
		RLFM	6.94	0.278	701.36
		FIGISS	12.44	0.006	795.70
		PFP-FM	26.12	8.763	233.04
		PFP-FM-CSA	30.95	3.078	1546.75
		FM-index	26.12	8.490	237.50
		Bowtie	7.09	0.94	28238.74
		Bowtie2	7.09	1.18	15242.00
SARS-100k	3,004,588,730	RLCSA	39.47	0.099	1690.22
		RLFM	25.01	0.571	1432.16
		FIGISS	25.57	0.009	1840.80
		PFP-FM	53.90	18.156	489.45
		PFP-FM-CSA	61.86	5.758	3150.72
		FM-index	51.85	16.73	434.55
		Bowtie	14.20	1.884	32143.48
		Bowtie2	14.19	2.37	33914.46
GRCh38	3,189,750,467	RLCSA	45.45	2.022	924.60
		RLFM	26.31	3.101	1839.25
		FIGISS	34.65	1.538	1440.19
		PFP-FM	71.13	37.862	1154.12
		FM-index	70.93	32.54	877.43
		PFP-FM-CSA	N/A	N/A	N/A
		Bowtie	13.99	1.833	2160.76
		Bowtie2	14.00	2.31	2170.32

The number of characters in each dataset (denoted as *n*) is in the second column

The construction time is reported in seconds (denoted as CONSTRUCT TIME)

The construction memory is reported in gigabytes (denoted as CONSTRUCT MEM)

The index size is reported in gigabytes (denoted as INDEX SIZE)

The implementation of the FM-index that we used was sourced from the `sdsl` library

suffix arrays are developed, they can be integrated in our method.

Results on human reference genome

After measuring the time and memory usage required to construct the data structure across all methods using the `GRCh38` dataset, we observed that `PFP-FM` exhibited the second most efficient construction time but used the most construction space (71 GB vs. 26 GB to 45 GB). More specifically, `PFP-FM` was able to construct the index between 1.25 and 1.6 times faster than the `FIGISS` and `RLFM`.

Next, we compare the performance of `PFP-FM` against other methods by performing 1000 `count` queries on, and illustrate the results in Fig. 7. Our findings demonstrate that `PFP-FM` consistently outperforms all other methods. The `RLCSA` method performs better than `RLFM` and `FIGISS` when the pattern length is over 125 but is still 3.9, 6.2, 3.4, and 7.1 times slower than `PFP-FM`. Meanwhile, the `RLFM` method exhibits a steady increase

in time usage, and it is 2.9, 14.2, 12.8, and 18.07 times slower than PFP-FM. It is worth noting that the FIGISS grammar is less efficient for non-repetitive datasets, as demonstrated in the research by Akagi et al. [25], which explains its (worse) performance on GRCh38 versus the SARS-100K dataset. Hence, FIGISS is 10.1, 25.5, 13.6, and 14.8 times slower than PFP-FM. These results are in line with the performance of our previous results, and demonstrate that PFP-FM has both competitive construction memory and time, and achieves a significant acceleration. Additionally, it is important to highlight that Bowtie exhibits higher efficiency in processing non-repetitive datasets. Despite its minimal memory requirements and smaller index size, Bowtie's processing is notably time-consuming.

Conclusion and future work

Hong et al. [26] recently gave a method for computing LZ77 parses quickly using PFP and, at least in theory, we can do the same. The key idea is that storing a data structure supporting range-minimum queries (RMQs) over the suffix array makes an FM-index for the string S partially persistent: to check whether a pattern Q occurs in $S[0..i-1]$ for some given i, we can search for Q in the FM-index for S, perform an RMQ over the suffix-array interval for Q, and check that the smallest value j such that $S[j..j+|Q|-1] = Q$ has $j+|Q|-1 \leq i-1$. In fact, if we store the FM-index for the reverse of S, use range-maximum queries instead of range-minimum queries, and check the suffix array after searching for every character of Q, then we can efficiently find the longest prefix of Q that occurs in $S[0..i-1]$. This allows us to compute efficiently the LZ77 parse of S. We are now working to compare how this approach compares to Hong et al.'s.

Notice that, when the query pattern Q starts and ends with trigger strings, we can perform the whole search in the FM-index for the parse P and need not use the FM-index for the string S at all. (See also related discussion in the arXiv preprint [27].) In fact, we are also now working to replace the FM-index for S by other data structures, in all cases. If Q starts with a trigger string but does not end with one, then instead of searching in the FM-index for S, we can find the lexicographic range of phrases in the dictionary D starting with the suffix of Q starting at the start of the rightmost trigger string. Once we have that range, we can begin the search in the FM-index for P with the corresponding range in the BWT of P. Finally, if Q neither starts nor ends with a trigger string, then we can use 2-dimensional range reporting on a grid with a point (x, y) whenever the co-lexicographically xth phrase in D appears in P before the lexicographically yth suffix of P (with the phrase and suffix overlapping at the trigger string). Specifically, we

1. find for the lexicographic range of phrases in D starting with the suffix of Q starting at the start of the rightmost trigger string,
2. start a search in the FM-index for P from the corresponding range in the BWT of P,
3. find the co-lexicographic range of phrases in D ending with the prefix of Q ending at the end of the leftmost trigger string,
4. use 2-dimensional range search on the grid to find all the substrings T of S in which the prefix of T ending at the end of T's leftmost trigger string matches the corresponding prefix of Q, and the suffix of T starting at the start of T's leftmost trigger string matches the corresponding suffix of Q — meaning T matches Q.

Author contributions

AH and MO implemented the software, downloaded the data, and ran all the experiments.CB and TG oversaw the software development and experimentsAH, MO, TG and CB drafted the paper.All authors helped write and edit the paper.

Funding

The funding details for the authors are as follows: 1. Aaron Hong and Marco Oliva: Supported by the NIH/NHGRI grant R01HG011392 to Ben Langmead. Supported by the NSF/BIO grant DBI-2029552 to Christina Boucher. 2. Dominik Köppl: Supported by the JSPS KAKENHI Grant Numbers JP21K17701, JP22H03551, and JP23H04378. 3. Hideo Bannai: Supported by the JSPS KAKENHI Grant Number JP20H04141. 4. Christina Boucher: Supported by the NIH/NHGRI grant R01HG011392 to Ben Langmead. Supported by the NSF/BIO grant DBI-2029552 to Christina Boucher. Supported by the NSF/SCH grant INT-2013998 to Christina Boucher. Supported by the NIH/NIAID Grant R01AI14180 to Christina Boucher. 5. Travis Gagie: Supported by the NIH/NHGRI grant R01HG011392 to Ben Langmead. Supported by the NSERC grant RGPIN-07185-2020 to Travis Gagie. Supported by the NSF/BIO grant DBI-2029552 to Christina Boucher.

Availability of data and materials

In our experiments, two datasets were employed: GRCH38 and SARS-CoV-2. The GRCH38 dataset can be accessed at https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.26/, while the SARS-CoV-2 dataset is available at https://www.ncbi.nlm.nih.gov/nuccore/1798174254.

Declarations

Competing interests

The authors declare no competing interests.

Received: 24 October 2023 Accepted: 12 March 2024
Published online: 10 April 2024

References

1. Ferragina P, Fischer J. Suffix arrays on words. In: Ma B, Zhang K, editors. Proceedings of the 18th Annual Symposium Combinatorial Pattern Matching (CPM). London: Springer; 2007. p. 328–39.

2. Deng J-J, Hon W-K, Köppl D. Sadakane K, FM-indexing grammars induced by suffix sorting for long patterns. In: Deng JJ, editor. Proceedings of the IEEE Data Compression Conference (DCC). Snowbird: IEEE; 2022. p. 63–72.
3. Ferragina P, Manzini G. Indexing compressed text. J ACM. 2005;52:552–81.
4. Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10(3):25–25.
5. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. Cornell Univ. 2013. https://doi.org/10.48550/arXiv.1303.3997.
6. Mäkinen V, Navarro G. Succinct suffix arrays based on run-length encoding. In: Apostolico A, Crochemore M, Park K, editors. Proceedings of the annual symposium on combinatorial pattern matching. Jeju Island: Springer; 2005. p. 45–56.
7. Gog S, Kärkkäinen J, Kempa D, Petri M, Puglisi SJ. Fixed block compression boosting in fm-indexes: theory and practice. Algorithmica. 2019;81:1370–91.
8. Gagie T, Navarro G, Prezza N. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J ACM. 2020;67(1):1–54.
9. Nunes DSN, Louza FA, Gog S, Ayala-Rincón M, Navarro G. A grammar compression algorithm based on induced suffix sorting. In: Nunes DSN, editor. 2018 Data Compression Conference. IEEE: Snowbird; 2018. p. 42–51. https://doi.org/10.1109/DCC.2018.00012.
10. Nong G, Zhang S, Chan WH. Two efficient algorithms for linear time suffix array construction. IEEE Trans Comput. 2011;60(10):1471–84. https://doi.org/10.1109/TC.2010.188.
11. Crescenzi P, Lungo AD, Grossi R, Lodi E, Pagli L, Rossi G. Text sparsification via local maxima. Theor Comput Sci. 2003;304(1–3):341–64.
12. Gog S, Beller T, Moffat A, Petri M. From theory to practice: plug and play with succinct data structures. In: Gog S, editor. Proceedings of the 13th symposium on experimental algorithms (SEA). Copenhagen: Springer; 2014. p. 326–37.
13. Siren J. Compressed suffix arrays for massive data. In: Karlgren J, Tarhio J, Hyyro H, editors. Proceedings of the 16th International Symposium String Processing and Information Retrieval (SPIRE). Berlin: Springer; 2009. p. 63–74.
14. Mäkinen V, Navarro G. Run-length FM-index. In: Proceedings of the DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later". 2004; pp. 17–19.
15. Manber U, Myers GW. Suffix arrays: a new method for on-line string searches. SIAM J Comput. 1993;22(5):935–48.
16. 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):13–11315.
17. Boucher C, Gagie T, Kuhnle A, Manzini G. Prefix-free parsing for building big BWTs. Algorithms Biol (WABI). 2018. https://doi.org/10.1186/s13015-019-0148-5.
18. Mun T, Kuhnle A, Boucher C, Gagie T, Langmead B, Manzini G. Matching reads to many genomes with the r-index. J Comput Biol. 2020;27(4):514–8. https://doi.org/10.1089/cmb.2019.0316.
19. Oliva M, Gagie T, Boucher C. Recursive prefix-free parsing for building big BWTs. In: Oliva M, editor. 2023 data compression conference (DCC). IEEE: Snowbird; 2023. p. 62–70.
20. Golynski A. Optimal lower bounds for rank and select indexes. 2006;387:370–81. https://doi.org/10.1007/11786986_33.
21. Mölder F, Jablonski KP, Letcher B, Hall MB, Tomkins-Tinch CH, Sochat V, Forster J, Lee S, Twardziok SO, Kanitz A, et al. Sustainable data analysis with Snakemake. F1000Research. 2021;10:33.
22. Langmead B, Wilks C, Antonescu V, Charles R. Scaling read aligners to hundreds of threads on general-purpose processors. Bioinformatics. 2018;35(3):421–32. https://doi.org/10.1093/bioinformatics/bty648.
23. Langmead B, Salzberg SL. Fast gapped-read alignment with Bowtie 2. Nat Methods. 2012;9(4):357–9.
24. Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10(3):25. https://doi.org/10.1186/gb-2009-10-3-r25.
25. Akagi T, Köppl D, Nakashima Y, Inenaga S, Bannai H, Takeda M. Grammar index by induced suffix sorting. In: Lecroq T, Touzet H, editors. Proceedings of the 28th international symposium on string processing and information retrieval (SPIRE). Cham: Springer; 2021. p. 85–99.
26. Hong A, Rossi M, Boucher C. LZ77 via prefix-free parsing. In: Hong A, editor. The proceedings of the symposium on algorithm engineering and experiments (ALENEX). Philadelphia: SIAM; 2023. p. 123–34.
27. Deng JJ, Hon W, Köppl D, Sadakane K. FM-indexing grammars induced by suffix sorting for long patterns. Snowbird: IEEE; 2021.

Publisher's Note