FM-Indexing Grammars Induced by Suffix Sorting for Long Patterns

Jin-Jie Deng*, Wing-Kai Hon*, Dominik Köppl†, and Kunihiko Sadakane‡

*Tsing Hua University
Hsinchu, Taiwan
jinjiedeng.jjd@gmail.com
wkhon@cs.nthu.edu.tw

† TMDU
Tokyo, Japan
koeppl.dsc@tmd.ac.jp

‡ The University of Tokyo
Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp

Abstract

The run-length compressed Burrows–Wheeler transform (RLBWT) used in conjunction with the backward search introduced in the FM index is the centerpiece of most compressed indexes working on highly-repetitive data sets like biological sequences. Compared to grammar indexes, the size of the RLBWT is often much bigger, but queries like counting the occurrences of long patterns can be done much faster than on any existing grammar index so far. In this paper, we combine the virtues of a grammar with the RLBWT by building the RLBWT on top of a special grammar based on induced suffix sorting. Our experiments reveal that our hybrid approach outperforms the classic RLBWT with respect to the index sizes, and with respect to query times on biological data sets for sufficiently long patterns, which could be interesting for aligning long reads in bioinformatics.

Keywords: BWT, grammar compression, count query

1 Introduction and Preliminaries

One prominent example of a text index is the FM-index [8]. It consists of a 2D geometric data structure built upon the BWT of the text, and can answer $\mathsf{count}(P)$, i.e., the number of occurrences of a pattern P in a text T, in time linear to the length of P multiplied by the operational cost of a 2D geometric data structure.

For constant alphabets, the FM-index achieves $\mathcal{O}(m)$ time for $\mathsf{count}(P)$ with $|P| = m$. It involves $\mathcal{O}(m)$ queries to the underlying 2D geometric data structure such as a wavelet tree, which answers the queries in a constant number of random accesses. However, in practical evaluations, we observe that these random accesses are a major bottleneck in the time performance of the FM-index. The BWT built on a grammar compressed string allows us to match non-terminals in one backward search step, hence allowing us to *jump over multiple characters* in one step, similar to the geometric BWT [3, Sect. 4].[1] Consequently, we spend less time on the cache-unfriendly wavelet tree, but more time (a) on extracting the grammar symbols stored in cache-friendly arrays, and (b) on a possibly slower traversal of the underlying wavelet tree due to the larger alphabet size encompassing non-terminals produced by the grammar. Our experiments reveal (a) that this extra work pays off for the reduced usage of the wavelet tree regarding the time performance, and (b), regarding the space, that the grammar captures the compressibility far better than the run-length compression of the BWT built on the plain text. Here, we leverage certain properties of the GCIS (grammar compression by induced suffix sorting) grammar [14], which have been discovered by Akagi et al. [1] and Díaz-Domínguez et al. [7] for determining non-terminals of the text matching portions of the pattern.

Definitions. With lg we denote the logarithm to base two (i.e., $\lg = \log_2$). Our computational model is the word RAM with machine word size $\Omega(\lg n)$, where n denotes the length of a given input string $T[1..n]$, which we call *the text*, whose characters are drawn from an integer alphabet $\Sigma = \{1, \ldots, \sigma\}$ of size $\sigma = n^{\mathcal{O}(1)}$. We call the elements of Σ *characters*. A *character run* is a maximal substring consisting of repetition of the same character. For a string $S \in \Sigma^*$, we denote with $S[i..]$ its i-th suffix, and with $|S|$ its length. The order $<$ on the alphabet Σ induces a lexicographic order on Σ^*, which we denote by $\prec$.

[1] A precursor is the research of Moffat and Isal [12] who studied the compression when treating words in natural languages as input characters of the BWT.

Given a character $\mathtt{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathrm{rank}_{\mathtt{c}}(j)$ counts the occurrences of $\mathtt{c}$ in $T[1..j]$, and the *select* query $T.\mathrm{select}_{\mathtt{c}}(j)$ gives the position of the j-th $\mathtt{c}$ in T. We stipulate that $\mathrm{rank}_{\mathtt{c}}(0) = \mathrm{select}_{\mathtt{c}}(0) = 0$.

Burrows–Wheeler Transform. The *BWT* of T is a permutation of the characters of $\tilde{T} := T\$$, where we appended an artificial character \$ smaller than all characters appearing in T. This BWT, denoted by BWT, is defined such that $\mathsf{BWT}[i]$ is the preceding character of $\tilde{T}$'s i-th lexicographically smallest suffix, or $\tilde{T}[|\tilde{T}|] = \$$ in case that this suffix is $\tilde{T}$ itself. Given a pattern $P[1..m]$, the *range* of $P[i..m]$ in BWT is an interval $[\ell_i..r_i]$ such that $\tilde{T}[j..]$ has $P[i..m]$ as a prefix if and only if $\tilde{T}[j..]$ is the k-th lexicographically smallest suffix with $k \in [\ell_i..r_i]$. The range $[\ell_i, r_i]$ of $P[i..m]$ can be computed from $P[i+1..m]$ by a backward search step on BWT with an array $C[1..\sigma]$, where $C[c]$ is the number of occurrences of those characters in BWT that are smaller than c, for $c \in [1..\sigma]$. Given the range of $P[i+1..m]$ is $[\ell_{i+1}..r_{i+1}]$, ℓ_i and r_i are determined by $\ell_i = C[P[i]] + 1 + \mathsf{BWT}.\mathrm{rank}_{P[i]}(\ell_{i+1})$ and $r_i = C[P[i]] + \mathsf{BWT}.\mathrm{rank}_{P[i]}(r_{i+1})$, with $\ell_m = C[P[m]] + 1$ and $r_m = C[P[m] + 1]$. We focus on ranges since the length of the range of P is $\mathsf{count}(P)$.

Grammar Compression Based on Induced Suffix Sorting. SAIS [13] is a linear-time algorithm for computing the suffix array. We briefly sketch the parts of SAIS needed for constructing our grammar. Starting with a text $T[1..n]$, we pad it with artificial characters # and \$ to its left and right ends, respectively, such that $T[0] = \#$ and $T[n+1] = \$$. We stipulate that $\# < \$ < c$ for each character $c \in \Sigma$. Central to SAIS is the type assignment to each suffix, which is either L or S: $T[i..]$ is an L suffix if $T[i..] \succ T[i+1..]$. Otherwise, $T[i..]$ is an S suffix, i.e., $T[i..] \prec T[i+1..]$, where we stipulate that $T[n+1] = \$$ is always type S. Since it is not possible that $T[i..] = T[i+1..]$, SAIS assigns each suffix a type. An S suffix $T[i..]$ is additionally an $\mathtt{S}^*$ suffix if $T[i-1..]$ is an L suffix. We further let $T[0..]$ be $\mathtt{S}^*$. The substring between two succeeding $\mathtt{S}^*$ suffixes is called an *LMS substring*. In other words, a substring $T[i..j]$ with $i < j$ is an LMS substring if and only if $T[i..]$ and $T[j..]$ are $\mathtt{S}^*$ suffixes and there is no $k \in [i+1..j-1]$ such that $T[k..]$ is an $\mathtt{S}^*$ suffix.

The LMS substrings induce a factorization of $T[0..n+1] = T_1 \cdots T_t$, where each factor starts with an LMS substring. We call this factorization *LMS factorization*. Next, we assign each factor the lexicographic rank of its respective LMS substring among all LMS substrings. Then, by replacing each factor with its rank, we obtain a string $T^{(1)}$ of these ranks. Then our grammar is defined as follows: The start symbol is X_T with the production rule $X_T \to T^{(1)}$, and the right-hand side of a non-terminal is an LMS substring without its last character, where the special characters # and \$ are omitted. We call this constructed grammar $\mathcal{G}_T$,[2] and let $\Sigma^{(1)}$ and $\sigma^{(1)}$ denote the set of symbols (i.e., non-terminals) of $T^{(1)}$ and its size, respectively. Further let $\pi(X) \in \Sigma^*$ be the right-hand side of a non-terminal $X \in \Sigma^{(1)}$.

Example 1.1 (Grammar Instance)**.** We build the grammar $\mathcal{G}_T$ on the example text $T :=$ `bacabacaacbcbc`. For that, we determine the types of all suffixes, which determine the LMS substrings:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
\$T# =	#	b	a	c	a	b	a	c	a	a	c	b	c	b	c	\$
	S*	L	S*	L	S*	L	S*	L	S*	S	L	S*	L	S*	L	S*
		D	C		B		C		A			E		E		

We obtain the grammar $\mathcal{G}_T$ with the following rules: A $\to$ aac, B $\to$ ab, C $\to$ ac, D $\to$ b, and E $\to$ bc.

The grammar has $\sigma^{(1)} := 5$ non-terminals on height 1. By replacing the LMS substrings with the respective non-terminals, we obtain the string $T^{(1)} :=$ DCBCAEE. In what follows, we study an approach that builds the BWT on this text, which is given by $\mathsf{BWT}^{(1)} :=$ ECCBD\$EA.

2 FM-Indexing the GCIS Grammar

The main idea of our index is that we build the same grammar $\mathcal{G}_P$ on P with start symbol $X_P \to P^{(1)}$ and translate the matching problem of P in T to matching $P^{(1)}$ in $T^{(1)}$. The problem is that the LMS factorization of P and the LMS factorization of the occurrences of P in T can look different since the occurrences of P in T are not surrounded by the artificial characters # and \$, but by different contexts of T. The question is whether there is a substring of $P^{(1)}$, for which we can be sure that each occurrence of P in T is represented in $T^{(1)}$ by a substring containing $P^{(1)}$. We call such a maximal substring a *core*, and use the following observation of Akagi et al. [1, Section 4.1]: Given an occurrence of P in T, the

[2] Note that the standard GCIS grammar [14] would recurse like SAIS on $T^{(1)}$ until the factorization is composed of a constant number of factors.

first LMS substring of P occurs as a suffix of an LMS substring of T that matches the beginning of this occurrence; the last LMS substring of P, up to a character run at its end, is represented as a prefix of an LMS substring of T matching the ending of this occurrence. The pattern matching is done in detail as follows.

2.1 Pattern Matching

Given a pattern P, we compute the first round of the GCIS grammar on P, which we call $\mathcal{G}_P$, where we use the same non-terminals as in $\mathcal{G}_T$ whenever their right-hand sides match. Then there are non-terminals $Y_1, \ldots, Y_p$ such that P has the LMS factorization $P = P_1 \cdots P_p$ with $P_y = \pi(Y_y)$ for each $y \in [1..p]$. According to the aforementioned observation of Akagi et al. each occurrence of P in T is captured by an occurrence of $Y_2 \cdots Y_{p-1}$ in $T^{(1)}$. So $Y_2, \ldots, Y_{p-1}$ do not only appear as non-terminals in the grammar of T, but they also appear as substrings in $T^{(1)}$ (if P occurs in T). In what follows, we call $Y_2, \ldots, Y_{p-1}$ the *core* of P, and show how to use the core to find P via $\mathsf{BWT}^{(1)}$, which is the BWT built on $T^{(1)}$.
If we turn $\mathsf{BWT}^{(1)}$ into an FM-index by representing it by a wavelet tree, it can find the core of P in $p-2$ backward search steps, i.e., returning an interval in the BWT that corresponds to all occurrences of $Y_2 \cdots Y_{p-1}$ in $T^{(1)}$, which corresponds to all occurrences of $P_2 \cdots P_{p-1}$ in T.
To find the missing suffix P_p, we proceed as follows. Let $\mathcal{R} := \{R_k\}_k \subset \Sigma^{(1)}$ be the set of all non-terminals R_k with P_p being a (not necessarily proper) prefix of $\pi(R_k)$. Since each $R_k \in \mathcal{R}$ received a rank according to the lexicographic order of its right-hand side, the elements in $\mathcal{R}$ form a consecutive interval in BWT, and this interval corresponds to occurrences of P_p. So starting with this interval the aforementioned backward search gives us the range for all occurrences of $P_2 \cdots P_p$.
However, this range may not contain *all* occurrences of $P_2 \cdots P_p$. That is because, according to the definition of a core, the rightmost non-terminal of a P occurrence may not cover P_p completely, but only $P_p[1..|P_p| - \ell - 1]$, where $P_p[|P_p| - \ell..|P_p|]$ is the longest character run that is a suffix of P_p, for $\ell \geq 0$. Now, suppose that the rule $X_p \to P_p[1..|P_p| - \ell]$ exists, then we need to check, for all non-terminals in the set $\mathcal{U} = \{U_j\}_j$ with $P_p[|P_p| - \ell..|P_p|]$ being a prefix of $\pi(U_j)$, whether $X_p U_j$ is a substring of T. With analogous reasoning, the occurrences of all elements of $\mathcal{U} \subset \Sigma^{(1)}$ form a consecutive range in BWT, and with a backward search from this range for X_p we obtain another range corresponding to P_p. However, this range combined with the range for $\mathcal{R}$ gives *all* occurrences of P_p. Consequently, if X_p exists, we need to perform the backward search not only for the range of $\mathcal{R}$, but also for $X_p\mathcal{U}$.

Example 2.1 (Pattern Matching)**.** Continuing with Example 1.1, let $P :=$ `cabaca` be a given pattern. The LMS factorization divides P into three factors P_1, P_2, and P_3. The core of P is `B`, evaluating to P_2 (see left figure). As previously explained, depending on the context of an occurrence of P in T, the last factor P_3 can be split at the last character run, which is $P[6]$. Indeed, there is no non-terminal in $\mathcal{G}_T$ having P_3 on its right hand side, but by splitting $P_3 = P[4..5]P[6] = P'_3 P'_4$ (middle figure), we find that P'_3 is equal to `C`, and the non-terminals `A`, `B`, and `C` have $P_4 =$ `a` as a prefix of their right-hand sides. These non-terminals form a consecutive interval $[2..5]$ in $\mathsf{BWT}^{(1)}$. With the backward search, we can find the interval of $P_2 P'_3 P'_4$ from $[2..5]$, as shown in the right of Fig. 1: From $[2..5]$, we match P'_3 corresponding to `C`, which gives the first and the second `C` in F, represented by the interval $[4..5]$. From there, we match P_2 corresponding to `B`, which gives the first `B` at position 3.

This concludes the explanation of our indexing data structure, with the omission of (a) how the right-hand sides of the non-terminals are indexed for providing fast lookups, and (b) how to compute $\mathsf{locate}(P)$ from the range of the occurrences of $P_2 \cdots P_p$. We treat both issues together with a practical improvement by limiting the lengths of these right-hand sides.

2.2 Limiting Factor Lengths

For practical performance, we introduce a *chunking parameter* $\lambda \in \mathcal{O}(\log_\sigma n)$. This parameter chops each LMS factor into factors of length λ with a possibly smaller last factor such that each non-terminal has a length of at most λ. The idea for such a small λ is that we can interpret the right-hand side of each non-terminal as an integer fitting into a constant number of machine words. For representing the right-hand sides of the non-terminals, we use compressed bit vectors B_{F} and B_{R}, each of length σ^λ. We represent $\pi(X)$ for each non-terminal as an integer $v \in [1..\sigma^\lambda]$ and store it by setting $B_{\mathrm{F}}[v] = 1$.

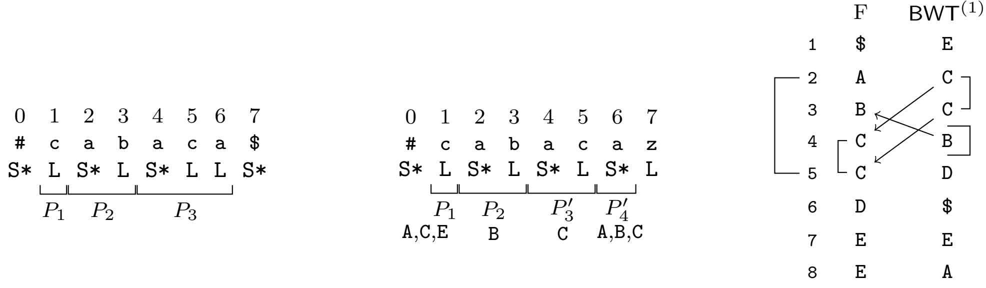

Figure 1: Matching the pattern $P =$ cabaca in $\mathsf{BWT}^{(1)}$ built on $T^{(1)} :=$ DCBCAEE with the algorithm described in Sect. 2.1. *Left:* Application of GCIS on P, analogously to Example 1.1 for the text, where we obtain three factors $P_1 = P[1]$, $P_2 = P[2..3]$ and $P_3 = P[4..6]$. *Middle:* Application of GCIS on Pz with an artificial character z larger than all other characters, where we obtain four factors $P = P_1 P_2 P'_3 P'_4$. (The factorization of the occurrence of P_3 within an occurrence of P in T is conditioned by the context of this occurrence within T.) *Right:* Matching $P_2 P'_3 P'_4$ in $\mathsf{BWT}^{(1)}$ with the backward search.

Similarly, we represent the reversed string of $\pi(X)$ as such an integer v' and set $B_{\mathrm{R}}[v'] = 1$. We endow B_{F} and B_{R} with rank/select-support data structures. We additionally store a permutation to convert a value of $B_{\mathrm{R}}.\mathrm{rank}_1$ to $B_{\mathrm{F}}.\mathrm{select}_1$.

Pattern Matching. Unfortunately, by limiting the right-hand sides of the non-terminals at length λ, the property that only the first and last non-terminal of the parsed pattern is not in the core no longer holds in general. Let again $P = P_1 \cdots P_p$ be the LMS factorization of our pattern. We assume that $p \geq 2$ and $|P| > \lambda$; the other cases are analyzed afterwards. For $x \in [2..p]$, we define the chunks $P_{x,1} \cdots P_{x,c_x} = P_x$ with $|P_{x,j}| = \lambda$ for each $j \in [1..c_x - 1]$ and $|P_{x,c_x}| \in [1..\lambda]$. Then, due to the construction of our chunks, there are non-terminals $Y_{x,j} \in \Sigma^{(1)}$ with $\pi(Y_{x,j}) = P_{x,j}$ for all $x \in [2..p-1]$ and $j \in [1..c_x]$. Hence, $Y_{2,1} \cdots Y_{2,c_2} Y_{3,1} \cdots Y_{p-1,1} \cdots Y_{p-1,c_{p-1}}$ is the core of P. The core can be found as a BWT range analogously as explained in Sect. 2.1.

Before searching the core, we first find P_p. We only analyze the case of an occurrence where the last character run in P_p is not part of the next factor in an occurrence in T. In that case, we find a range of non-terminals whose right-hand sides start with P_{p,c_p}. In detail, we interpret P_{p,c_p} as a binary integer v having $|P_{p,c_p}| \lg \sigma$ bits. Then we create two integers v_1, v_2 by padding v with '0' and '1' bits to v's right end (interpreting the right end as the bits encoding the end of the string P_{p,c_p}), respectively, such that v_1 and v_2 have $\lambda \lg \sigma$ bits with $v_1 \leq v_2$. This gives us the range of ranks $[B_{\mathrm{F}}.\mathrm{rank}_1(v_1)..B_{\mathrm{F}}.\mathrm{rank}_1(v_2)]$ of all non-terminals whose right-hand sides start with P_{p,c_p}, and this interval of ranks translates to a range in $\mathsf{BWT}^{(1)}$. Because we know that P_p was always a prefix of a non-terminal in Sect. 2.1, we can apply the backward search to extend this range to the range of $P_{p,1} \cdots P_{p,c_p}$, and then continue with searching the core.

Finally, to extend this range to the full pattern, we remember that an occurrence of P_1 in T was always a suffix of the right-hand side of a non-terminal. First suppose that $|P_1| > \lambda$. Then such a former right-hand side has been chunked into strings of length λ, where the last string has a length in $[1..\lambda]$. Because we want to match a suffix, we have therefore λ different ways in how to chunk $P_1 = P_{1,1} \cdots P_{1,c_1}$ into the same way with $|P_{1,c_1}| \in [1..\lambda]$. Let us fix one of these chunkings. We try to extend the range of the core by $P_{1,2} \cdots P_{1,c_1}$ with the backward search steps as before. If we successfully obtain a range, then we use the bit vector B_{R} and interpret the reverse of $P_{1,1}$ like P_{p,c_p} above as an integer to obtain an interval I of colexicographic ranks for all non-terminals whose reversed right-hand sides have the reverse of $P_{1,1}$ as a prefix (i.e., whose right-hand sides have $P_{1,1}$ as a suffix). Finally, we use the permutation from B_{R} to B_{F} for each non-terminal of the interval I, and locate each non-terminal in the wavelet tree individually. If $|P_1| \leq \lambda$, then we can proceed analogously, treating P_1 like $P_{1,1}$.

Small Patterns. Here, we accommodate patterns with $p = 1$ or $|P| < \lambda$, First, for $p = 1$ but $|P| \geq \lambda$, we have $P = P_1$, and we treat P_1 exactly like in the above algorithm by trying λ different chunkings $P_1 = P_{1,1} \cdots P_{1,c_1}$ with $|P_{1,c_1}| \in [1..\lambda]$, find all non-terminals having P_{1,c_1} as prefixes of their right-hand sides, extend the matching interval to an interval of $P_{1,2} \cdots P_{1,c_1}$ via backwards search steps, and finally use the colexicographic rankings of B_{R} to find $P_{1,1} \cdots P_{1,c_1}$.

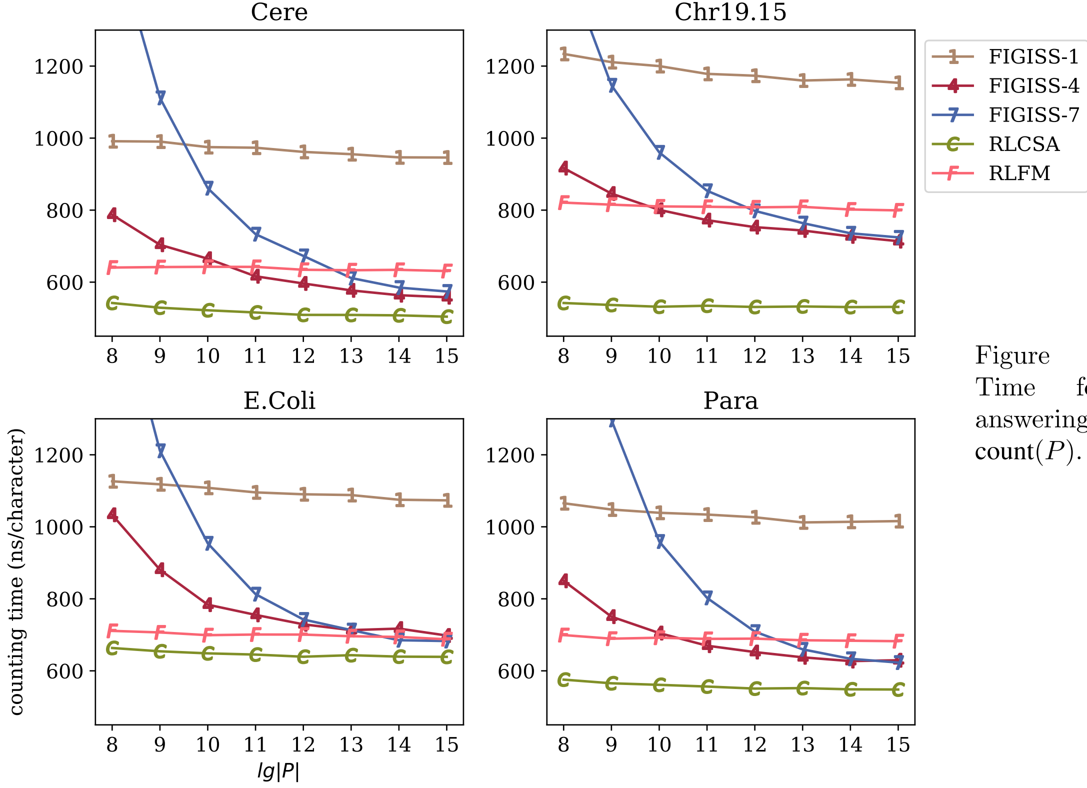

Figure 2: Time for answering count(P).

For $|P| < \lambda$, we need a different data structure: We create a generalized suffix tree on the right-hand sides of all non-terminals. The *string label* of a node v is the concatenation of edge labels read from the root to v. We augment each node by the number of occurrences of its string label in T. For a given pattern P, we find the highest node v whose string label has P as a prefix. Then the answer to count(P) is the stored number of occurrences in v. For the implementation, we represent the generalized suffix tree in LOUDS, and store the occurrences in a plain array by the level order induced by LOUDS.

Table 1: Comparison of $\mathsf{RLFM}^{(0)}$ and FIGISS on the datasets described in Sect. 3. The space is in Mebibytes ([MiB]), and '[M]' denotes mega (10^6). $r^{(0)}$ and $r^{(1)}$ are the number of character runs in $\mathsf{BWT}^{(0)}$ and $\mathsf{BWT}^{(1)}$, respectively, and $\sigma^{(0)}$ and $\sigma^{(1)}$ are, respectively, the number of their different symbols. The column $\lg|P|$ is the logarithmic pattern length at which FIGISS starts to become faster than $\mathsf{RLFM}^{(0)}$ on answering count(P). See Sect. 2.2 for a description of the chunking parameter λ.

input text			$\mathsf{RLFM}^{(0)}$		FIGISS				
name	space [MiB]	σ	$r^{(0)}$ [M]	space [MiB]	λ	space [MiB]	$\sigma^{(1)}$	$r^{(1)}$ [M]	$\lg|P|$
CERE	439.9	6	11.6	26.8	1	26.5	6	11.6	-
					4	17.3	271	5.8	11
					7	14.9	1790	5.0	13
CHR19.15	845.8	6	32.3	70.8	1	69.7	6	32.3	-
					4	47.1	140	16.5	9
					7	39.7	1174	13.8	12
E.COLI	107.5	16	15.0	26.2	1	25.4	16	15.0	-
					4	17.8	809	7.3	13
					7	15.1	2356	6.2	13
PARA	409.4	6	15.6	34.4	1	34.0	6	15.6	-
					4	22.6	296	7.9	11
					7	19.4	2701	6.7	13

Table 2: Final index space, construction peak memory, and construction time.

text	index	final space [MiB]	memory peak [GiB]	time [s]
CERE	faster-minuter	20.7	6.8	110.5
	FIGISS-4	17.3	5.7	285.5
	FIGISS-7	**14.9**	**3.7**	263.9
	RLCSA	33.4	7.7	239.0
	RLFM$^{(0)}$	26.8	6.8	**102.2**
CHR19.15	faster-minuter	50.2	13.6	221.1
	FIGISS-4	47.1	11.1	629.7
	FIGISS-7	**39.7**	**7.3**	565.8
	RLCSA	88.2	15.1	663.2
	RLFM$^{(0)}$	70.8	13.6	**206.3**
E.COLI	faster-minuter	20.1	1.7	25.8
	FIGISS-4	17.8	1.4	70.3
	FIGISS-7	**15.1**	**1.0**	65.0
	RLCSA	26.6	1.9	51.2
	RLFM$^{(0)}$	26.2	1.7	**24.4**
PARA	faster-minuter	25.5	6.6	105.1
	FIGISS-4	22.6	5.5	254.6
	FIGISS-7	**19.4**	**3.6**	236.9
	RLCSA	40.4	6.5	208.7
	RLFM$^{(0)}$	34.4	6.6	**97.8**

3 Implementation and Evaluation

Central to our implementation[3] is the wavelet tree implementation built upon the run-length compressed $\mathsf{BWT}^{(1)}$, for which we used the class `sdsl::wt_rlmn`. This class is a wrapper around the actual wavelet tree to make it usable for the RLBWT. Therefore, it is parameterized by a wavelet tree implementation, which we set to `sdsl::wt_ap`, an implementation of the alphabet-partitioned wavelet tree of Barbay et al. [2]. Since we only care about answering count, we do neither sample the suffix array nor its inverse. The bit vectors B_{F} and B_{R} are realized by the class `sdsl::sd_vector<>` leveraging Elias–Fano compression.

Evaluation Environment. We evaluated all our experiments on a machine with Intel Xeon E3-1231v3 clocked at 3.4GHz running Ubuntu 20.04.2 LTS. The used compiler was g++ 9.3.0 with compile options `-std=c++17 -O3`.

Datasets. We set our focus on DNA sequences, for which we included the datasets CERE, ESCHERICHIA_COLI (abbreviated to E.COLI), and PARA from the repetitive corpus of Pizza&Chili[4]. We additionally stored 15 of 1000 individual sequences of the human chromosome 19[5] in the dataset CHR19.15. For the experiments we assume that all texts use the byte alphabet. In a preprocessing step, after reading an input text T, we reduce the byte alphabet to an alphabet $\Sigma = \{1, \ldots, \sigma\}$ such that each character of Σ appears in T. For technical reasons, we further assume that the texts end with a null byte (at least the used classes in the sdsl need this assumption), which is included in the alphabet sizes σ of our datasets. We present the characteristics of our datasets in Table 1 in the first columns.

Experiments. In the following experiments, we call our solution FIGISS-λ, evaluate it for the chunking parameters $\lambda \in \{1, 4, 7\}$ (cf. Sect. 2.2), and compare it with the FM-index $\mathsf{RLFM}^{(0)}$ built on $\mathsf{BWT}^{(0)}$ (the BWT of T) run-length compressed, again without any sampling.[6]

Note that the sampling is only useful for locate queries, and therefore would be only a memory burden in our setting. While FIGISS uses `sdsl::wt_ap` suitable for larger alphabet sizes, $\mathsf{RLFM}^{(0)}$ uses `sdsl::wt_huff`, a wavelet tree implementation optimized for byte alphabets. Table 1 shows the space requirements of $\mathsf{RLFM}^{(0)}$ and FIGISS, which are measured by the serialization framework of sdsl. There, we observe that the larger λ gets, the better FIGISS compresses. However, we are pessimistic that this

[3] Our implementation is written in C++17 using the sdsl-lite library [9]. The code is available at `https://github.com/jamie-jjd/figiss`.

[4] http://pizzachili.dcc.uchile.cl/repcorpus/real

[5] http://dolomit.cs.tu-dortmund.de/tudocomp/chr19.1000.fa.xz

[6] A more thorough evaluation is found in the full version of this paper [6].

will be strictly the case for $\lambda > 7$ since the introduced number of symbols exponentially increases while the number of runs $r^{(1)}$ approaches a saturation curve. The case $\lambda = 1$ can be understood as a baseline: Here, the right-hand sides of all terminals are single characters. Hence, this approach does not profit from any benefits of our proposed techniques, and is provided to measure the overhead of our additional computation (e.g., the dictionary lookups). While FIGISS-4 is faster, it uses more space than FIGISS-7. Compared to $\mathsf{RLFM}^{(0)}$, FIGISS always uses less space, and for the majority of values of λ, answering $\mathsf{count}(P)$ is faster for sufficiently long lengths $|P|$, which can be observed in the plots of Fig. 2. There, we measure the time for $\mathsf{count}(P)$ with $|P| = 2^x$ for each $x \in [8..15]$. For each data point and each dataset T, we extract 2^{12} random samples of equal length from T, perform the query for each sample, and measure the average time per character.[7] Here, we can empirically assess that the larger λ is, the steeper the falling slope of the average query time per character is for short patterns. That is because of the split of P_1 into λ different chunkings. Interestingly, it seems that the used wavelet tree variant `sdsl::wt_ap` uses less memory than `sdsl::wt_huff` used for $\mathsf{RLFM}^{(0)}$ regarding the space comparison of FIGISS-1 and $\mathsf{RLFM}^{(0)}$ in Table 1. Additionally, we evaluate the index data structures RLCSA [15], faster-minuter [10], and the implementation [4] of the lz-index [11]. faster-minuter and lz-index are always the fastest and the slowest solutions, respectively, for pattern matching, and are thus omitted in Fig. 2. We also conducted experiments with a plain FM-index `sdsl::csa_wt<sdsl::wt_blcd<>>`, which is competitive with faster-minuter regarding the measured times, but does not compress.
Finally, in Table 2, we evaluated the index construction with respect to time, peak memory, and the final index size. We observe that FIGISS has the smallest index size and the lowest memory requirements. A drawback is the slower construction time, which is however still competitive with RLCSA. lz-index is omitted; it has construction times as worse as FIGISS-1. Its construction memory peak is competitive with FIGISS-7, but it has larger space requirements for the index.

4 Future Work

The chunking into substrings of length λ is rather naive. Running a locality sensitive grammar compressor like ESP [5] on the LMS substrings will produce factors of length three with the property that substrings are factorized in the same way, except maybe at their borders. Thus, we expect that employing a locality sensitive grammar will reduce the number of symbols and therefore improve $r^{(1)}$. We further want to parallelize our implementation, and strive to beat $\mathsf{RLFM}^{(0)}$ for smaller pattern lengths. Also, we would like to conduct our experiments on larger datasets like sequences usually maintained by pangenome indexes of large scale.

Acknowledgements This work was supported by JSPS KAKENHI grant numbers JP21K17701 and JP21H05847.

References

[1] T. Akagi, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Grammar index by induced suffix sorting. In *Proc. SPIRE*, volume 12944 of *LNCS*, pages 85–99, 2021.
[2] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.
[3] Y. Chien, W. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Geometric BWT: compressed text indexing via sparse suffixes and range searching. *Algorithmica*, 71(2):258–278, 2015.
[4] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016.
[5] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.
[6] J. J. Deng, W. Hon, D. Köppl, and K. Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. *ArXiv CoRR*, abs/2110.01181, 2021.
[7] D. Díaz-Domínguez, G. Navarro, and A. Pacheco. An LMS-based grammar self-index with local consistency properties. In *Proc. SPIRE*, volume 12944 of *LNCS*, pages 100–113, 2021.

[7] We extract the patterns from the input such that we can be sure that each pattern actually exists. Non-existing patterns would give FIGISS an advantage since finding the first factor $P_{1,1}$ takes a significant amount of time.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[9] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, volume 8504 of *LNCS*, pages 326–337, 2014.

[10] S. Gog, J. Kärkkäinen, D. Kempa, M. Petri, and S. J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.

[11] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483: 115–133, 2013.

[12] A. Moffat and R. Y. K. Isal. Word-based text compression using the Burrows-Wheeler transform. *Inf. Process. Manag.*, 41(5):1175–1192, 2005.

[13] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[14] D. S. N. Nunes, F. A. da Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. DCC*, pages 42–51, 2018.

[15] J. Sirén. Compressed suffix arrays for massive data. In *Proc. SPIRE*, volume 5721 of *LNCS*, pages 63–74, 2009.