Faster run-length compressed suffix arrays

Nathaniel K. Brown
Department of Computer Science, Johns Hopkins University, USA

Travis Gagie[1]
Faculty of Computer Science, Dalhousie University, Canada

Giovanni Manzini
Department of Computer Science, University of Pisa, Italy

Gonzalo Navarro
Department of Computer Science, University of Chile, Chile

Marinella Sciortino
Department of Mathematics and Computer Science, University of Palermo, Italy

Abstract

We first review how we can store a run-length compressed suffix array (RLCSA) for a text T of length n over an alphabet of size σ whose Burrows-Wheeler Transform (BWT) consists of r runs in $O\left(r\log(n/r) + r\log\sigma + \sigma\right)$ bits such that later, given character a and the suffix-array (SA) interval for P, we can find the SA interval for aP in $O(\log r_a + \log\log n)$ time, where r_a is the number of runs of copies of a in the BWT. We then show how to modify the RLCSA such that we find the SA interval for aP in only $O(\log r_a)$ time, without increasing its asymptotic space bound. Our key idea is applying a result by Nishimoto and Tabei (ICALP 2021) and then replacing rank queries on sparse bitvectors by a constant number of select queries. We also review two-level indexing and discuss how our faster RLCSA may be useful in improving it. Finally, we briefly discuss how two-level indexing may speed up a recent heuristic for finding maximal exact matches of a pattern with respect to an indexed text.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Run-length compressed suffix arrays, interpolative coding, two-level indexing

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.10

Funding *Nathaniel K. Brown*: Supported in part by a Johns Hopkins University Computer Science PhD Fellowship and NIH grants R21HG013433 and R01HG011392.
Travis Gagie: Funded in part by NSERC grant RGPIN-07185-2020.
Giovanni Manzini: Partially supported by the NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem (Spoke 6, CUP: I53C22000780001) and by the project PAN-HUB funded by the Italian Ministry of Health (ID: T4-AN-07, CUP: I53C22001300001)
Gonzalo Navarro: Funded in part by CeBiB under Basal Funds FB0001 and AFB240001, ANID, Chile.
Marinella Sciortino: Partially supported by the project "ACoMPA" (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem (Spoke 6) and by MUR PRIN project "PINC" (no. 2022YRB97K)

Acknowledgements Many thanks to the first author's CSCI 4419 / 6106 students Anas Alhadi, Nour Allam, Dove Begleiter, Nithin Bharathi Kabilan Karpagavalli, Suchith Sridhar Khajjayam and Hamza Wahed, and to Christina Boucher, Ben Langmead, Jouni Sirén and Mohsen Zakeri.

[1] Corresponding author.

1 Introduction

Grossi and Vitter's compressed suffix arrays (CSAs) [11] and Ferragina and Manzini's FM-indexes [8] are sometimes treated as almost interchangeable, but their query-time bounds are quite different. With a CSA for a text T of length n over an alphabet of size σ, when given a character a and the suffix-array (SA) interval for a pattern P we can find the SA interval for aP in $O(\log n_a)$ time, where n_a is the number of occurrences of a in the text; with an FM-index we use $O(\log \sigma)$ time. This difference carries over to run-length compressed suffix arrays (RLCSAs) [18, 24] and run-length compressed FM-indexes (RLFM-indexes) [10, 17], with both taking space proportional to the number r of runs in the Burrows-Wheeler Transform (BWT) of the text but the former being generally faster for texts over large alphabets with relatively few runs of each character, and the latter being faster for texts over smaller alphabets.

In Section 2 we review (with some artistic license) CSAs and RLCSAs. In Subsection 3 we show how to use interpolative coding to build an RLCSA for T that takes $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits and allows us to find the SA interval for aP from that of P in $O(\log r_a + \log \log n)$ time, where r_a is the number of those runs in the BWT containing copies of a. In Subsection 3.2 we review a result by Nishimoto and Tabei [20] about splitting the runs in the BWT so that we can evaluate LF in constant time, without increasing the number of runs by more than a constant factor. In Subsection 3.3 we present our main result: how to modify the RLCSA from Section 2 such that finding the SA interval for aP takes only $O(\log r_a)$ time, without increasing the asymptotic space bound. In Section 4 we discuss two-level indexing, for which we build one index for the text and another for the parse of the text, and how our faster RLCSA may be more suitable for indexing parses than current options. Finally, in Section 5 we briefly discuss how two-level indexing may speed up a recent heuristic for finding long maximal exact matches (MEMs) of a pattern with respect to an indexed text.

2 Preliminaries

Suppose we are given a text $T[0..n-1]$ over an alphabet of size σ and asked to index it such that, given a pattern $P[0..m-1]$, we can quickly count the number of occurrences of P in T. More specifically, we want to find the interval in the suffix array (SA) of T containing the starting positions of occurrences of P. Consider the matrix whose rows are the lexicographically sorted cyclic shifts of T and let F and L be the first and last column of that matrix, respectively; this means F contains the characters in T in lexicographic order and L is the BWT of T.

2.1 Compressed suffix arrays

The key idea behind compressed suffix arrays (CSAs) is to store $\Psi[0..n-1]$ compactly while supporting certain searches on it quickly, where $\Psi[0..n-1]$ is the permutation of $\{0, \ldots, n-1\}$ such that $\Psi[i]$ is the position of SA entry $(\mathrm{SA}[i]+1) \bmod n$ in $\mathrm{SA}[0..n-1]$ or, equivalently, the position in L of $F[i]$. (This means Ψ is the inverse of the LF mapping used in FM-indexes.) By the definition, Ψ consists of at most σ increasing intervals — one for each distinct character that occurs in the text, corresponding to the interval of suffixes starting with that character — and if we can support fast binary searches on these intervals then we can support fast pattern matching.

For example, consider the text

$$T = \texttt{CCTGGGCGAT\$CTTACACGAT\$GTTACCAGCT\$CTTACGCGCT\$CTGACGAATT\$CTTACGCGAT\#}\,,$$

for which SA, Ψ, F and L are shown on the left in Figure 1. If we know SA[22..28] is the SA interval for `CG` (in the green rectangle) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi[36..48] = 6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$$

for `G` (in the red rectangle, with Ψ values between 22 and 28 shown as orange arrows and the others shown as black arrows) for the successor $\Psi[41] = 23$ of 22 and the predecessor $\Psi[43] = 28$ of 28. We thus learn that the SA interval for `GCG` is SA[41..43] (in the blue rectangle). Knowing this, we can continue backward stepping.

2.2 Run-length compressed suffix arrays revisited

Run-length compressed suffix array (RLCSA) were introduced in [24] for indexing highly repetitive collections. In this section we present an alternative, but functionally equivalent, description of RLCSAs which is more suitable for describing our improvements.

▶ **Definition 1.** *For a text $T[0..n-1]$, the array $L'[0..r-1]$ stores the sequence of r characters in the runs of the run-length encoding of L.*

▶ **Definition 2.** *For a text $T[0..n-1]$, the array $F'[0..r-1]$ stores the r characters in L' in lexicographic order.*

▶ **Definition 3.** *For a text $T[0..n-1]$, the array $\Psi'[0..r-1]$ is the permutation of $\{0,\ldots,r-1\}$ such that $\Psi'[i]$ is the position of $F'[i]$ in L'.*

In this paper we view a RLCSA as a data structure storing $\Psi'[0..r-1]$ compactly while supporting certain searches on it quickly. By the definition of Ψ', it still consists of at most σ increasing intervals — one for each distinct character that occurs in T, corresponding to the interval of suffixes starting with that character — and if we can still support fast binary searches on these intervals then we can still support fast pattern matching.

For example, consider

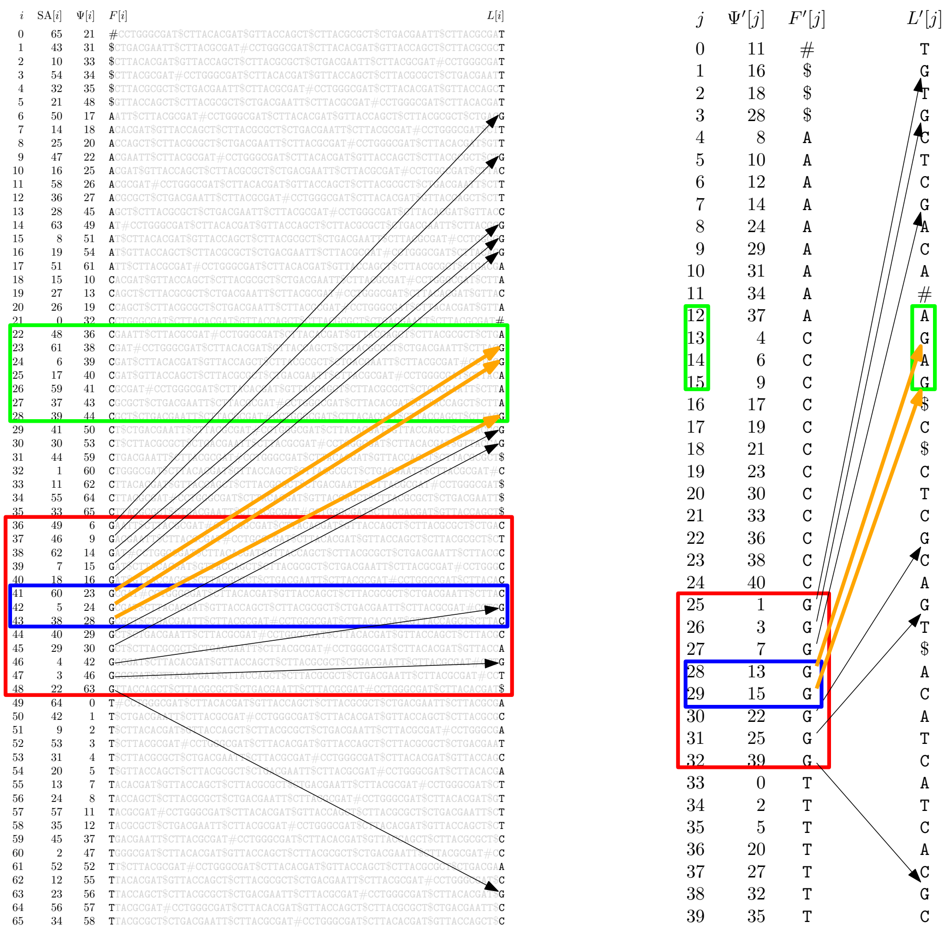
$$T = \texttt{CCTGGGCGAT\$CTTACACGAT\$GTTACCAGCT\$CTTACGCGCT\$CTGACGAATT\$CTTACGCGAT\#}$$

again, for which Ψ', F' and L' are shown on the right in Figure 1. If we know the SA interval SA[22..28] for `CG` starts at offset 0 in the L run of character $L'[12]$ and ends at offset 1 in the L run of character $L'[15]$ (in the green rectangle) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi'[25..33] = 1, 3, 7, 13, 15, 22, 25, 39$$

for `G` (in the red rectangle, with Ψ' values between 12 and 15 shown as orange arrows and the others shown as black arrows) for the successor $\Psi'[28] = 13$ of 12 and the predecessor $\Psi'[29] = 15$ of 15.

Because L' and F' do not have the predecessor-successor relationship of L and F, we cannot deduce that the SA interval for `GCG` starts in the L run of character $L'[28]$ and ends in the L run of character $L'[29]$ (and, in fact, in this example it does not). Instead, we store two n-bit SD-bitvectors [21], B_L and B_F, with r copies of 1 each. The 1s in B_L mark the



i	SA[i]	Ψ[i]	$F[i]$		$L[i]$
0	65	21	#	CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGA	T
1	43	31	$	CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGC	T
2	10	33	$	CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGA	T
3	54	34	$	CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAAT	T
4	32	35	$	CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGC	T
5	21	48	$	GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGA	T
6	50	17	A	ATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGAC	G
7	14	18	A	CACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CT	T
8	25	20	A	CCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GT	T
9	47	22	A	CGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CT	G
10	16	25	A	CGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTA	C
11	58	26	A	CGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CT	T
12	36	27	A	CGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CT	T
13	28	45	A	GCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTAC	C
14	63	49	A	T#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGC	G
15	8	51	A	T$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGC	G
16	19	54	A	T$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACAC	G
17	51	61	A	TT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACG	A
18	15	10	C	ACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTT	A
19	27	13	C	AGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTA	C
20	26	19	C	CAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTT	A
21	0	32	C	CTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT	#
22	48	36	C	GAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTG	A
23	61	38	C	GAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTAC	G
24	6	39	C	GAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGG	G
25	17	40	C	GAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTAC	A
26	59	41	C	GCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTT	A
27	37	43	C	GCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTT	A
28	39	44	C	GCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTAC	G
29	41	50	C	T$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGC	G
30	30	53	C	T$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCA	G
31	44	59	C	TGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT	$
32	1	60	C	TGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#	C
33	11	62	C	TTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT	$
34	55	64	C	TTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT	$
35	33	65	C	TTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT	$
36	49	6	G	AATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGA	C
37	46	9	G	ACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$C	T
38	62	14	G	AT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACG	C
39	7	15	G	AT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGG	C
40	18	16	G	AT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACA	C
41	60	23	G	CGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTA	C
42	5	24	G	CGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTG	G
43	38	28	G	CGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTA	C
44	40	29	G	CT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACG	C
45	29	30	G	CT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACC	A
46	4	42	G	GCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCT	G
47	3	46	G	GGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CC	T
48	22	63	G	TTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT	$
49	64	0	T	#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCG	A
50	42	1	T	$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCG	C
51	9	2	T	$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCG	A
52	53	3	T	$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAA	T
53	31	4	T	$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAG	C
54	20	5	T	$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACG	A
55	13	7	T	ACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$C	T
56	24	8	T	ACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$G	T
57	57	11	T	ACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$C	T
58	35	12	T	ACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$C	T
59	45	37	T	GACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$	C
60	2	47	T	GGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#C	C
61	52	52	T	T$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGA	A
62	12	55	T	TACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$	C
63	23	56	T	TACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$	G
64	56	57	T	TACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$	C
65	34	58	T	TACGCGCT$CTGACGAATT$CTTACGCGAT#CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$	C

j	$\Psi'[j]$	$F'[j]$	$L'[j]$
0	11	#	T
1	16	$	G
2	18	$	T
3	28	$	G
4	8	A	C
5	10	A	T
6	12	A	C
7	14	A	G
8	24	A	A
9	29	A	C
10	31	A	A
11	34	A	#
12	37	A	A
13	4	C	G
14	6	C	A
15	9	C	G
16	17	C	$
17	19	C	C
18	21	C	$
19	23	C	C
20	30	C	T
21	33	C	C
22	36	C	G
23	38	C	C
24	40	C	A
25	1	G	G
26	3	G	T
27	7	G	$
28	13	G	A
29	15	G	C
30	22	G	A
31	25	G	T
32	39	G	C
33	0	T	A
34	2	T	T
35	5	T	C
36	20	T	A
37	27	T	C
38	32	T	G
39	35	T	C

Figure 1 For

$T =$ `CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#`

we show SA, Ψ, F and L on the left and the Ψ′, F' and L' on the right. If we know SA[22..28] is the SA interval for `CG` (in the green rectangle on the left) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi[36..48] = 6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$$

for `G` (in the red rectangle on the left, with Ψ values between 22 and 28 shown as orange arrows and the others shown as black arrows) for the successor $\Psi[41] = 23$ of 22 and the predecessor $\Psi[43] = 28$ of 28. We thus learn that the SA interval for `GCG` is SA[41..43] (in the blue rectangle on the left).

On the other hand, if we know SA[22..28] starts at offset 0 in the L run of character $L'[12]$ — that is, at offset 0 in the 13th run, counting from 1 — and ends at offset 1 in the L run of character $L'[15]$ (in the green rectangle on the right), then we can search in the increasing interval

$$\Psi'[25..32] = 1, 3, 7, 13, 15, 22, 25, 39$$

for `G` (in the red rectangle, with Ψ′ values between 12 and 15 shown as orange arrows and the others shown as black arrows) for the successor $\Psi'[28] = 13$ of 12 and the predecessor $\Psi'[29] = 15$ of 15 (in the blue rectangle on the right). We then use select and rank queries on two n-bit sparse vectors to find the SA interval for `GCG`, the L runs containing that interval's starting and ending positions, and those positions' offsets in those runs.

starting positions of runs in L and the 1s in B_F mark the positions in F of the marked characters in L. In our example

$$B_F = \texttt{11100110111001111111111100010111011011100}\,\mathbf{101}\,\texttt{0011110000010101111000}$$
$$B_L = \texttt{10000011011101100101111101001001110011100}\,\mathbf{011}\,\texttt{0111111111110001011110}\,.$$

The interval $B_F[41..43]$ in B_F starting immediately before the bit with offset 0 in the block whose starting position is marked with the 29th copy of 1 and ending immediately before the bit with offset 1 in the block whose starting position is marked with the 30th copy of 1, is shown in blue. (We are interested in the blocks marked with the 29th and 30th copies of 1 because we count from 0 in the j column in Figure 1, so those blocks correspond to $\Psi'[28]$ and $\Psi'[29]$.) We can find this interval with 2 select_1 queries on B_F, which take constant time.

The corresponding interval $B_L[41..43]$ in B_L is also shown in blue, starting immediately before the bit with offset 3 in the block whose starting position is marked with the 22nd copy of 1 and ending immediately before the bit with offset 1 in the block whose starting position is marked with the 24th copy of 1. We can find the 2 indices 22 and 24 with 2 rank_1 queries on B_L, which take $O(\log\log n)$ time. This means the SA interval for `GCG` is $\text{SA}[41..43]$ and it starts at offset 3 in the L run of character $L'[21]$ and ends at offset 1 in the L run of character $L'[23]$. Knowing this we can continue backward stepping.

The RLCSA in Sirén's PhD thesis [24] for a text $T[0..n-1]$ with r BWT runs takes $O\left(r\log(n/r) + r\log\sigma + \sigma\log n\right)$ bits. Given a character a and the SA interval for P, it can find the SA interval for aP in $O(\log n)$ time.

3 Faster RLCSAs

3.1 Searchable Interpolative coding

Suppose we are given an increasing list $\ell_1, \ldots, \ell_k$ of k integers in the range $[0..n-1]$. To encode them with *interpolative coding* [19], we first write $\ell_{\lceil k/2\rceil}$ using $\lfloor\lg(n-1)\rfloor + 1$ bits (except that we write 0 using 1 bit). All the numbers $\ell_1, \ldots, \ell_{\lceil k/2\rceil - 1}$ are in the range $[0..\ell_{\lceil k/2\rceil} - 1]$, so we can encode them recursively. All the numbers $\ell_{\lceil k/2\rceil + 1}, \ldots, \ell_k$ are in the range $[\ell_{\lceil k/2\rceil} + 1..n-1]$, so we can encode them recursively as $\ell_{\lceil k/2\rceil+1} - \ell_{\lceil k/2\rceil} - 1, \ldots, \ell_k - \ell_{\lceil k/2\rceil} - 1$. Each encoding has $O(\log n)$ bits, so we can read them in $O(1)$ time. If we imagine the list stored as keys in a balanced binary search tree then we encode the keys according to a pre-order traversal: when we reach each key ℓ_i, we know ℓ_i lies between the numbers shown to the left and right of ℓ_i and we encode ℓ_i using the maximum number of bits we would need for any key in that range.

For example, if $n = 66$, $k = 13$ and the list is $6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$ then, as illustrated in Figure 2, we start by encoding $\ell_7 = 24$ using $\lfloor\lg 65\rfloor + 1 = 7$ bits as `0011000`. We then encode $\ell_3 = 14$ using $\lfloor\lg 23\rfloor + 1 = 5$ bits as `01110`. We then encode $\ell_1, \ell_2, \ell_5, \ell_4, \ell_6 = 6, 9, 16, 15, 23$ as `0110`, `010`, `0001`, `0`, `110`, and $\ell_{10}, \ell_8, \ell_9, \ell_{12}, \ell_{11}, \ell_{13}$ as `000101`, `011`, `0`, `001111`, `1011`, `10000`. When we reach 46, say, in a pre-order traversal of the tree in Figure 2, we know it lies between 31 and 65, so we encode it using $\lfloor\lg(65-31)\rfloor + 1 = 6$ bits as $(46-31)_2 = \texttt{001111}$.

The binary search tree has height $\lfloor\lg k\rfloor$ and the bottom level contains at most k keys. By Jensen's Inequality, we encode those keys using $O(k\log(n/k) + k)$ bits. Similarly, there at most $k/2^h$ keys at height h and we encode those keys using $O\left(\frac{k}{2^h}\log\frac{n}{k/2^h} + \frac{k}{2^h}\right) =$

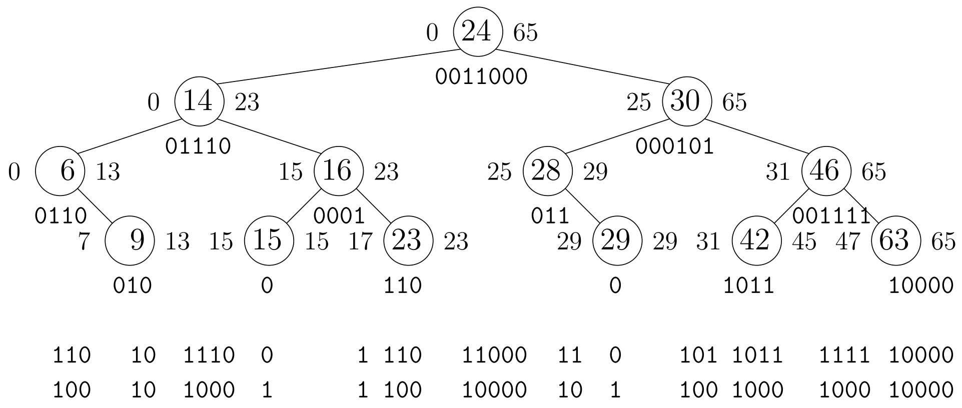


Figure 2 A balanced binary search tree storing the $k = 13$ keys from the increasing list $6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$ with each key in the range $[0..n - 1 = 65]$. When we reach each key in a pre-order traversal or binary search, we know it lies between the two values shown to its left and right, so we can encode it as the binary number shown below it, using a total of $O(k \log(n/k) + k)$ bits. If we store a bitvector marking the start of each encoding as visited in an in-order traversal, as shown below the tree, then we can omit the leading 0s from the encodings and support binary search in time $O(\log k)$ without changing our asymptotic space bound.

$O\left(\frac{k}{2^h} \log(n/k) + \frac{k(h+1)}{2^h}\right)$ bits. Since

$$\sum_{h=0}^{\lfloor \lg k \rfloor} \frac{k(h+1)}{2^h} = O(k)\,,$$

we use $O(k \log(n/k) + k)$ bits in total.

In this paper we want to perform binary search on the list — the reader may have noticed that our example is $\Psi[36..48]$ from Figure 1 — so we want fast access to the encodings of the numbers in it in the order we check them in a binary search. We can store the encodings according to an in-order traversal instead of a pre-order traversal, and store an uncompressed bitvector with as many bits as there are in the concatenation of the encodings and 1s marking where the encodings start. Since the bitvector delimits the encodings, however, we can delete the leading 0s from each encoding before concatenating them and building the bitvector. The in-order encodings for our example are shown below the tree in Figure 2, with the leading 0s removed, and the bitvector is shown below them. Since the bitvector uses at most as many bits as the encodings, we still use $O(k \log(n/k) + k)$ bits in total and — although random access still takes $O(\log k)$ time — we can perform binary search in $O(\log k)$ total time. This scheme is similar to Teuhola's [25] and Claude, Nicholson and Seco's [5].

To find the successor of 22 in the list, we start at the root knowing $n = 66$ and $k = 13$ and perform $\text{select}_1(7)$ and $\text{select}_1(8)$ queries on the bitvector to find the starting and ending positions of the encoding `0011000` of $\ell_7 = 24$ in the range $[0..65]$. Since $22 < 24$, we then perform $\text{select}_1(3)$ and $\text{select}_1(4)$ queries to find the starting and ending positions of the encoding `01110` of $\ell_3 = 14$ in the range $[0..23]$. Since $22 > 14$, we then perform $\text{select}_1(5)$ and $\text{select}_1(6)$ queries to find the starting and ending positions of the encoding `0001` of $\ell_5 = 16$ in the range $[15..23]$. Since $22 > 16$, we then perform $\text{select}_1(6)$ and $\text{select}_1(7)$ queries to find the starting and ending positions of the encoding `110` of $\ell_6 = 23$ in the range $[17..23]$. Since $22 < 23$, we know the successor of 22 in L is 23. We can find the predecessor of 28 in $O(\log k)$ time symmetrically.

If we apply interpolative coding with fast binary search to the increasing interval of Ψ for a character a in a text T of length n, then we use $O(n_a \log(n/n_a) + n_a)$ bits and can support binary search in $O(\log n_a)$ time, where n_a is the frequency of a in T. If we do this for all the characters then we use $O(n(H_0(T) + 1))$ bits, where H is the 0th-order empirical entropy of T. If we encode the increasing interval of Ψ' for a with interpolative coding, then we use $O(r \log(r/r_a) + r_a)$ bits and can support binary search in $O(\log r_a)$ time, where r_a is the number of runs of copies of a in the BWT of T (and, equivalently, in L). If we do this for all the characters then we use $O(r(H_0(L') + 1))$ bits, where L' is again the sequence of r characters in the runs of the run-length encoding of T. To be able to find the increasing interval for a in Ψ', we store an r-bit uncompressed bitvector with 1s marking where the intervals start.

▶ **Theorem 4.** *We can store Ψ' for T in $O(r(H_0(L') + 1)) \subseteq O(r \log \sigma)$ bits and support binary search in the increasing interval for a character a in $O(\log r_a)$ time, where r_a is the number of runs of copies of a in the BWT of T.*

To use Theorem 4 in an RLCSA, we store
- an uncompressed bitvector marking which distinct characters occur in T, in $O(\sigma)$ bits;
- the SD-vectors B_F and B_L in $O(r \log(n/r))$ bits;
- an uncompressed bitvector with 1s marking where the intervals for the characters start in Ψ', in $O(r)$ bits;
- Ψ' in $O(r \log \sigma)$ bits.

If we are given a and the SA interval for P then we can find the SA interval for aP by
- using a rank query on the first uncompressed bitvector to find a's rank among the distinct characters that occur in T, in $O(1)$ time;
- using rank queries on B_L to find the runs in L overlapping the SA interval for P, in $O(\log \log n)$ time;
- using select queries on the second uncompressed bitvector to find the interval for a in Ψ', in $O(1)$ time;
- using binary search in the interval for a in Ψ' to find the successor and predecessor of the ranks of the first and last runs in L overlapping the SA interval for P, in $O(\log r_a)$ time;
- using select queries on B_F and arithmetic to find the SA interval for aP in $O(1)$ time.

We store $O(r \log(n/r) + r \log \sigma + \sigma)$ bits in total and find the SA interval for aP in $O(\log r_a + \log \log n)$ total time. Notice that the $O(\log \log n)$ term in the query time comes only from the rank query on B_L.

▶ **Corollary 5.** *We can store an RLCSA for T in $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits such that, given character a and the SA interval for P, we can find the SA interval for aP in $O(\log r_a + \log \log n)$ time.*

3.2 Splitting Theorem for RLCSAs

Nishimoto and Tabei [20] showed how we can split the runs in L such that no block in B_F overlaps more than a constant number of blocks in B_L without increasing the number of runs by more than a constant factor, and then store LF in $O(r \log n)$ bits and evaluate it in constant time. Brown, Gagie and Rossi [4] slightly generalized their key theorem:

▶ **Theorem 6** (Nishimoto and Tabei [20]; Brown, Gagie and Rossi [4]). *Let π be a permutation on $\{0, \ldots, n-1\}$,*

$$P = \{0\} \cup \{i \ : \ 0 < i \leq n-1, \pi(i) \neq \pi(i-1) + 1\},$$

and $Q = \{\pi(i) \; : \; i \in P\}$*. For any integer* $d \geq 2$*, we can construct* P' *with* $P \subseteq P' \subseteq \{0, \ldots, n-1\}$ *and* $Q' = \{\pi(i) \; : \; i \in P'\}$ *such that*

- *if* $q, q' \in Q'$ *and* q *is the predecessor of* q' *in* Q'*, then* $|[q, q') \cap P'| < 2d$*,*
- $|P'| \leq \frac{d|P|}{d-1}$*.*

If $L[i] = L[i-1]$ then $\mathrm{LF}(i) = \mathrm{LF}(i-1) + 1$, so

$$\{0\} \cup \{i \; : \; 0 < i \leq n-1, \mathrm{LF}(i) \neq \mathrm{LF}(i-1) + 1\}$$

has cardinality r. If $\mathrm{LF}(i) = \mathrm{LF}(i-1) + 1$ then, since Ψ and LF are inverse permutations, $\Psi[j] = \Psi[j-1] + 1$ where $j = \mathrm{LF}(i)$. Therefore,

$$\{0\} \cup \{j \; : \; 0 < j \leq n-1, \Psi[j] \neq \Psi[j-1] + 1\}$$

also has cardinality r and applying Theorem 6 with $d = 2$ to Ψ splits the runs in the BWT such that no block in B_F overlaps more than 3 blocks in B_L, without increasing the number of runs by more than a factor of 3/2. In our example, the number of runs increases by only 1, from 40 to 41, as shown below with the split block — corresponding to the first run of 6 copies of `T` in L — in red:

$$\begin{array}{rcl} B_F & = & \texttt{11100110111001111111111110001011101101110010100111}\mathbf{100100}\texttt{1010111000} \\ B_L & = & \mathbf{100100}\texttt{110111011001011111010010011100111000110111111111110001011110}\,. \end{array}$$

Suppose we apply Theorem 6 with $d = 2$ to Ψ and then store, for $0 \leq b < r$, the index of the block in B_L containing $\mathrm{LF}(i_b)$ and $\mathrm{LF}(i_b)$'s offset in that block, where i_b is the starting position of block b in B_L. Nishimoto and Tabei called this the *move table* for LF (see also [4, 26]) and it takes a total of $O(r \log n)$ bits. If we know $B_L[j]$ is in block b in B_L with offset $j - i_b$ then, since the block in B_F to which LF maps block b in B_L now overlaps at most the block containing $B_L[\mathrm{LF}(i_b)]$ and the next 2 blocks in B_L, we can find the index of the block in B_L containing $B_L[\mathrm{LF}(j)] = B_L[\mathrm{LF}(i_b)] + j - i_b$ and $B_L[\mathrm{LF}(j)]$'s offset in that block with at most 2 constant-time select queries on B_L. We could use at most 2 constant-time lookups instead if we have the starting positions of the blocks in B_L stored explicitly in another $O(r \log n)$ bits.

3.3 A faster RLCSA without rank queries

Recall that the $O(\log \log n)$ term in the query-time bound in Corollary 5 comes only from the use of rank queries on an SD-vector. Since rank and select queries can be combined to support predecessor queries and select queries on sparse bitvectors can easily be supported in constant time and space polynomial in the number of 1s, rank queries on compact sparse bitvectors inherit lower bounds from predecessor queries [3] — so they cannot be implemented in constant time. Therefore, to get rid of that $O(\log \log n)$ term, we must somehow avoid rank queries.

We could replace the rank queries with a move table, but that would result in an $O(r \log n)$ term in our space bound. Instead, we introduce an uncompressed $2r$-bit bitvector B_{FL} indicating how the starting positions of the blocks in F and L are interleaved. Specifically, we scan B_F and B_L simultaneously — assuming we have already applied Theorem 6 to them so that no block in F overlaps more than 3 blocks in L (so r is a constant factor larger than it was before the application of the theorem) — and

- if we see 0s in both bitvectors in position i then we write nothing;
- if we see a 1 in B_F and a 0 in B_L then we write a 1 (indicating that a block starts in F);

- if we see a 0 in B_F and a 1 in B_L then we write a 0 (indicating that a block starts in L);
- if we see 1s in both bitvectors then we write a 0 and then a 1 (indicating that blocks start in both L and F).

This way, $B_{FL}.\mathrm{select}_1(j)$ tells us which at most 3 blocks in L — those corresponding to the 0 preceding the jth copy of 1 in B_{FL} and possibly to the next 2 copies of 0 — could overlap block j in F (counting from 1). We can then find the starting positions of those blocks in L using at most 3 select queries on B_L.

For our example, taking B_F and B_L to be as shown just after Theorem 6,

```
          0123456789012345678901234567890123456789012345 67890123456789012345
  B_F  =  1110011011100111111111110001011101101110010100 11110010010101111000
  B_L  =  100100110111011001011111010010011100111000110111111111110001011110
```

(with the grey numbers only to show positions), we have

```
            0123456789012345678901234567890123456789
  B_FL  =   0111010101010100101110110101010101010110...
                 012345678901234 567890123456789012345678901
              ...1001101010110010010101010001000110110101 00.
```

In position 38 we see 1s in both B_F and B_L, so we write 01 in B_{FL} (in positions 49 and 50, respectively); in positions 39 and 40 we see 0s in both in B_F and B_L, so we write nothing; in position 41 we see a 1 in B_F and a 0 in B_L, so we write a 1 in B_{FL}; in position 42 we see a 0 in B_F and a 1 in B_L, so we write a 0 in B_{FL}; in position 43 we see 1s in both B_F and B_L, so we write 01 in B_{FL}; in position 44 we see 0s in both B_F and B_L, so we write nothing; and in position 45 we see a 0 in B_F and a 1 in B_L, so we write a 0 in B_{FL} (in position 55). Admittedly, when $n = 66$ and after applying Theorem 6 $2r = 82$, it seems foolish to store a $2r$-bit uncompressed bitvector instead of simply storing B_L uncompressed. This is due to the small size of our example, however; for massive and highly repetitive datasets, r can easily be hundreds of times smaller than n.

Suppose we know the SA interval SA[41..43] for aP starts at offset 0 in block 28 in F and ends at offset 1 in block 29 in F and we want to find which blocks contain its starting and ending positions in L and the offsets of those positions. In Section 2, we performed 2 rank queries on B_L, but now we perform queries $B_{FL}.\mathrm{select}_1(29) = 51$ and $B_{FL}.\mathrm{select}_1(30) = 54$ (with arguments 29 and 30 instead of 28 and 29 because we mark with a 1 the starting of the first block in F, which we index with 0; the results 51 and 54 are indexed from 0 as well). Since the 29th and 30th copies of 1 are $B_{FL}[51]$ and $B_{FL}[54]$ (shown in red above), they are preceded by the $51 - 29 + 1 = 23$rd and $54 - 30 + 1 = 25$th copies of 0, respectively.

Because we applied Theorem 6, this means the 29th and 30th blocks in F (shown in red in B_F above) overlap the 23rd block in L and possibly the 24th and 25th blocks (shown in blue in B_L), and the 25th block and possibly the 26th and 27th blocks (also shown in blue in B_L). Notice that, because we split the 34th block in F but the first block in L for Theorem 6, the block numbers we find in F are the same as in Section 2 but the block numbers we find in L will be incremented. Although in general we need 6 select queries on B_L, in this case we can use only 5 — $B_L.\mathrm{select}_1(23), \ldots, B_L.\mathrm{select}_1(27)$ — to find where these blocks begin in constant time, and determine which contain the starting and ending positions of the SA interval SA[41..43]: the 23rd and the 25th, respectively.

In short, we replace a rank query on SD-bitvector B_L by queries on uncompressed bitvector B_{FL} and constant-time select queries on B_L. This gives us the following theorem:

▶ **Theorem 7.** *We can store an RLCSA for T in $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits such that, given character a and the SA interval for P, we can find the SA interval for aP in $O(\log r_a)$ time, where r_a is the number of runs of copies of a in the BWT of T.*

Instead of viewing B_{FL} as replacing slow rank queries while using the overall same space, we can also view it (and B_F and B_L) as replacing an $O(r \log n)$-bit move table while using the same overall query time. Brown, Gagie and Rossi [4] implemented a similar approach to speeding up LF computations in an RLFM-index, but only alluded to it briefly in their paper — the path to `Bitvector` in their Figure 3 — and gave no analysis nor bounds. We conjecture that a similar approach can also be applied to reduce the size of fast move tables for ϕ and ϕ^{-1} [13], which return $\mathrm{SA}[i-1]$ and $\mathrm{SA}[i+1]$ when given $\mathrm{SA}[i]$.

4 Two-level indexing

Corollary 5 and Theorem 7 suggest that RLCSAs should perform well compared to FM-indexes and RLFM-indexes when the BWT is over a fairly large alphabet and the number of runs of each character is fairly small; Ordóñez, Navarro and Brisaboa [23] have confirmed this experimentally. When indexing a highly repetitive text over a small alphabet, we can make RLCSAs more practical by storing a table of k-tuples that tells us in which range of Ψ' to search based on which character we are trying to match and which $k-1$ characters we have just matched. (This table can be represented with a bitvector to save space.) The table for our example from Figure 1 and $k = 2$ is shown below:

`#C`	0	`AG`	8	`CT`	20..24	`T#`	33
`$C`	1..2	`AT`	9..12	`GA`	25..27	`T$`	33
`$G`	3	`CA`	13..14	`GC`	28..29	`TA`	34..35
`AA`	4	`CC`	15..16	`GG`	30..31	`TG`	36..37
`AC`	4..7	`CG`	17..19	`GT`	32	`TT`	38..39

This says that if we want the SA interval for `GCG` and we have just matched the suffix `CG`, then we should search in the range $\Psi'[28..29]$. On the other hand, notice that the largest range of Ψ' in which we will ever search is now $\Psi'[20..24]$ — of length 5 — when we are trying to match a `C` after just matching a `T`; without such a table, the largest range we search is $\Psi'[13..24]$ — of length 12 — when trying to match a `C`.

There are interesting cases in which we want to index highly repetitive texts over large alphabets, however. For example, consider indexing a minimizer digest of a pangenome — considering minimizers as meta-characters from a large alphabet instead of tuples of characters from a small alphabet [1, 2, 7, 27] — or *two-level indexing* such a text. For two-level indexing we build one index for the text and another for a parse of the text; the alphabet of the parse is the dictionary of distinct phrases, which is usually large, but the parse itself is usually much smaller than the text and its BWT is usually still run-length compressible (albeit less than the BWT of the text) when the text is highly repetitive.

Something like two-level indexing was proposed by Deng, Hon, Köppl and Sadakane [6] but they did not use an index for the text and its absence made their implementation quite slow for all but very long patterns. Hong, Oliva, Köppl, Bannai, Boucher and Gagie [12] described another approach, which we will review here, but they used standard FM-indexes for the text and the parse instead of RLFM-indexes, so their two-level index was noticeably faster but hundreds of times larger than its competitors.

Consider the 50 similar toy genomes of length 50 each in Figure 3. Suppose we parse their concatenation similarly to `rsync`, by inserting a phrase break whenever we see a trigger string

```
CTTCCGCGGTGATAAAGGGGGCGGTAATGTCGCGAAACAGTCTTTTCTA$     CTTACGCGGTGATTCAGGGGGCGGTAATTTCGCGGATCAGTCTTTTCTA$
CTTACGCGGTGATACAGGGGGCCGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACGCGGTTATCCAGGGGGTGGTACTTTCGGTGAACAGTCTGTTCTA$
CTTACGCGACGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACGCGGTGATCCAGGGGGCAGTAATTTCGCGGAACAGTCTTTTCTA$
CTTATGCGATGATCCTGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACGCGATGATCCAGGGGGCGGTAATTTCGCTGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACACTCTTCTCTA$     CTTACGCGATGATCCATTGGGCGGTAATTCCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGTGGGCGGTCTTTTCGCGGAACAGTCTTTTCGA$     CTTACGCGATGATCCATGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGCAACAGTCTTTTCTA$     CTTACGCGATGATCCAGGGGGCTGTATTTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCTCGGAACAGTCTTTTCTA$     CTTACGCGCTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTATGCGGTGATCCACGGGGCGGAAGTATCGCGGAACAGTCTTTTTTA$     CTTACGAGATGAGCTAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAACTTCGCGGAACAGTCTTTTCTA$     CTTACGCGATGCTCCAGGGGGCGGTGATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGACGATCCAGGGGGCAGTAATTTCGCGGAACAGTCTTTTCTA$     CTTTCGCGATGATCCAGGGGGCGGTCATTTCGCGGAACAGTCTTTTCTA$
CATACGCGGGGATCCAAGGGGCGGTAATTTCGCGGAACAGTCTTTGACA$     CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGCACAGTCCTTTCTA$
CTTTCACGGTGATCCAGGGGTGGGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$     CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACTCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTT$
CTTACGCGGTGATCCAGGGGGCGCTAATTTCGCGAAACAGTCTTTTCTA$     CTTACGCGATGATCCAGAGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGTGGTGATCCAGGGGGCGGTAATTTCGAGGAACAGTCTTTAATA$     CTTACGCGGTGATCCAGGGGGCGGTAATTCCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGCGCGGTAATTTCGCGGAACAATCTTTTCTA$     CTTACGCGGGGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTAATCTA$     CTTACGCGGTGATCCAGGGGTCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$     CATACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$     CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$     CCTACGCGATGATGCAGGGGGCGGTAATTTCGCGGAACAGTCTTTCCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTGCGCGGAACAGTCTTTTCTA$     CTTACGCGATGTTCCAGGGGGCGGTAATTTCGCGGAATAGTTTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGGGGAACAGTCTTTTCTA$     CTTACGCGGTGATCCAGCGGGCGGTAATTTCGCGGAATAGTCTTTTCTA$
CTTACGCGATCTTCCAGGGGGCCGAAATTTCGCGTAACAGTCTTTTCTA$     CTTACGCGGTAATCCAGGGGGCGGTAATGTCGCGGAACAGTCTTTTCTA#
```

Figure 3 A set of 50 similar toy genomes of length 50 each, with the first 49 separated by copies of `$` and the last one terminated by `#`.

— `ACA`, `ACG`, `CGC`, `CGG`, `GAC`, `GAG`, `GAT`, `GTG`, `GTT`, `TCG` or `TCT` — or when we reach the `TA#` at the end of the text. (Considering $\# = \$ = 0$, $\texttt{A} = 1$, $\texttt{C} = 2$, $\texttt{G} = 3$ and $\texttt{T} = 4$ and viewing triples as 3-digit numbers in base 5, the trigger strings are the triples in the concatenation whose values are congruent to 0 modulo 6.) If we replace each phrase in the parse by its lexicographic rank in the dictionary of distinct phrases, counting from 1, and terminate the sequence with a 0, we get the 562-number sequence shown in Figure 4. With a larger example, of course, we obtain longer phrases on average and better compression from the parsing.

Run-length compression naturally works better on the BWT of the concatenation of the genomes than on the BWT of the parse, as shown in Figures 5 and 6. Again, with a larger example we would achieve better compression, also from the run-length compressed BWT (RLBWT) of the parse. Even this small example, however, gives some intuition how the dictionary of distinct phrases in the parse is usually large, but the parse is usually much smaller than the text and its BWT is usually still run-length compressible. In this case there are 90 distinct phrases in the dictionary, the parse is less than a quarter as long as the text, and the average run length in its RLBWT is slightly more than 2. An FM-index based on the RLBWT of the parse would generally use at least about $\lceil \lg 90 \rceil = 7$ rank queries on bitvectors for each backward step. The most common value in the runs, 19, occurs in only 16 runs, so we should spend at most about $\lg 16 = 4$ steps in each binary search.

To search for a pattern, we start by backward stepping in the index for the text until we reach the left end of the rightmost trigger string in the pattern. We keep count of how often each trigger string occurs in the text and an n-bit sparse bitvector with 1s marking the lexicographic ranks of the lexicographically least suffixes starting with each trigger string. This way, when we reach the left end of the rightmost trigger string, with a rank query on that bitvector we can compute the lexicographic ranks of the suffixes starting with the suffix of the pattern we have processed so far among all the suffixes starting with trigger strings, and map from the index of the text into the index for the parse. The width of the BWT interval stays the same and may span several lexicographically consecutive phrases in the

```
44, 55, 79, 19, 11, 70, 22, 46, 64, 88,  6, 22, 55, 79, 19, 17, 59,  22, 55, 12,
64, 88,  6, 22, 48, 45, 19, 32, 73, 22, 55, 12, 64, 88,  8, 50, 39,  73, 22, 55,
12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 43, 78, 41,  6, 622, 50, 50,
36, 58, 78, 87, 22, 55, 12, 64, 87,  6, 22, 55, 79, 19, 32, 73, 22,  51, 12, 64,
88,  6, 22, 55, 79, 19, 32, 74, 40, 45, 12, 64, 88,  9, 79, 19, 26,  45, 58, 13,
22, 55, 12, 64, 90, 22, 50, 50, 32, 68, 22, 55, 12, 64, 88,  6, 22,  48, 45, 19,
30, 22, 55, 12, 64, 88,  4, 22, 55, 57, 25, 73, 22, 55, 12, 64, 86,   2,  1, 45,
79, 19, 35, 60, 22, 55, 12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22,  55, 12, 21,
88,  6, 22, 50, 50, 32, 73, 22, 55, 12, 64, 88,  6, 22, 55, 79, 19,  31, 73, 22,
46, 64, 88,  6, 79, 65, 19, 32, 73, 18, 47, 64, 83, 22, 55, 79, 19,  29, 55, 73,
22, 55, 12, 21, 88,  6, 22, 50, 50, 32, 73, 22, 55, 12, 64, 16,  6,  22, 55, 79,
19, 32, 73, 22, 55, 12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22, 55,  12, 64, 88,
 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 21, 88,  6, 22, 50, 50, 32,  72, 55, 12,
64, 88,  6, 22, 55, 79, 19, 32, 73, 45, 56, 64, 88,  6, 22, 50, 41,  77, 22, 61,
64, 88,  6, 22, 55, 79, 19, 75, 73, 22, 55, 19, 24, 88,  6, 22, 55,  82, 20, 62,
45, 79, 12, 64, 66, 41,  6, 22, 55, 79, 19, 30, 22, 55, 12, 64, 88,   6, 22, 50,
50, 32, 73, 22, 80, 64, 88,  6, 22, 50, 50, 38, 71, 55, 12, 64, 88,   6, 22, 50,
50, 37, 73, 22, 55, 12, 64, 88,  6, 22, 50, 50, 33, 22, 55, 12, 64,  88,  6, 22,
51, 81, 32, 73, 22, 55, 12, 64, 88,  6, 18, 19, 49, 42, 73, 22, 55,  12, 64, 88,
 6, 22, 50, 54, 79, 19, 85, 22, 55, 12, 64, 88, 10, 22, 50, 50, 32,  76, 22, 55,
12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22, 55, 23, 63,  6, 22, 55,  79, 19, 32,
73, 22, 55, 12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 64,  88,  7, 45,
79, 19, 32, 73, 22, 55, 12, 64, 88, 67, 22, 50, 50, 27, 58, 73, 22,  55, 12, 64,
88,  6, 22, 55, 79, 19, 32, 71, 55, 12, 64, 88,  6, 22, 55, 57, 32,  73, 22, 55,
12, 21, 88,  6, 22, 55, 79, 19, 34, 45, 73, 22, 55, 12, 64, 88,  4,  22, 55, 79,
19, 32, 73, 22, 55, 12, 64, 88,  6, 22, 55, 79, 19, 32, 73, 22, 55,  12, 64, 88,
 5, 22, 50, 50, 52, 73, 22, 55, 12, 64, 84, 22, 50, 66, 32, 73, 22,  55, 15, 89,
 6, 22, 55, 79, 19, 28, 53, 73, 22, 55, 14, 88,  6, 22, 55, 69, 70,  22, 55, 12,
64, 88,  3,  0
```

Figure 4 The 563-number sequence (20 numbers per line) over the alphabet $\{0, \dots, 90\}$ we get from the concatenation of the toy genomes in Figure 3 by parsing, replacing each phrase by its rank in the dictionary (counting from 1) and appending a 0.

A^{8} T^{1} A^{41} T^{28} G^{1} T^{18} C^{1} T^{1} G^{2} T^{1} G^{27} A^{1} G^{10} C^{1} T^{1} A^{1} G^{6} T^{1} C^{1}

A^{1} G^{1} T^{1} G^{2} T^{2} C^{4} T^{39} A^{1} T^{3} G^{1} A^{5} T^{1} C^{1} A^{41} T^{1} G^{2} T^{42} C^{2} T^{2} C^{1}

A^{1} T^{1} C^{1} G^{1} C^{1} G^{2} C^{1} G^{1} A^{1} C^{6} A^{1} C^{13} T^{1} C^{22} A^{1} C^{22} T^{1} C^{22} T^{1} A^{1}

G^{2} C^{2} A^{2} G^{10} A^{1} G^{25} T^{1} G^{6} T^{1} A^{1} G^{1} A^{4} G^{15} T^{2} G^{1} C^{1} A^{2} G^{2} A^{3} C^{1}

A^{27} C^{1} A^{2} G^{1} A^{9} T^{1} A^{1} G^{1} C^{1} A^{4} G^{1} C^{1} T^{1} A^{1} C^{6} A^{1} C^{12} G^{1} C^{11} T^{1}

C^{10} G^{2} A^{35} T^{1} A^{7} C^{1} $\$^{2}$ C^{2} T^{45} G^{1} T^{3} G^{1} T^{1} $\$^{1}$ T^{3} G^{2} C^{1} G^{2} A^{1} T^{1}

A^{2} G^{17} T^{2} A^{9} T^{1} A^{7} T^{1} A^{1} T^{18} C^{1} T^{2} C^{1} T^{9} G^{1} T^{9} A^{3} G^{1} C^{1} A^{22} T^{1}

G^{1} T^{1} G^{35} T^{1} G^{9} T^{1} G^{2} A^{1} G^{17} T^{1} G^{23} T^{1} G^{18} T^{1} G^{4} A^{1} G^{4} C^{1} A^{1} T^{17}

C^{1} T^{28} G^{1} C^{1} G^{2} T^{1} A^{1} T^{1} A^{1} G^{2} C^{1} G^{1} T^{2} $\$^{44}$ T^{1} $\#^{1}$ A^{1} T^{2} $\$^{1}$ T^{1}

$\$^{1}$ A^{1} C^{1} T^{44} C^{4} G^{6} T^{1} G^{15} T^{1} G^{22} T^{1} C^{3} T^{1} C^{1} A^{1} T^{2} G^{2} T^{4} C^{1} T^{9}

G^{1} T^{10} C^{1} T^{13} C^{1} A^{1} C^{13} T^{1} C^{2} T^{2} C^{1} G^{1} T^{1} G^{4} C^{16} T^{1} C^{4} T^{1} G^{2} C^{38}

G^{1} C^{4} A^{1} C^{2} G^{1} C^{1} G^{8} C^{1} G^{29} C^{2} T^{1} C^{20} G^{1} C^{3} A^{1} T^{1} C^{2} G^{1} C^{6} A^{1}

C^{10} G^{1} C^{26} G^{2} C^{1} G^{54} A^{1} G^{5} T^{1} G^{16} A^{1} G^{8} C^{1} G^{7} T^{1} G^{2} C^{3} G^{5} C^{1} G^{13}

A^{1} G^{2} T^{1} G^{19} A^{23} T^{1} A^{18} G^{1} T^{1} G^{3} C^{25} G^{1} C^{14} T^{1} C^{1} G^{1} C^{10} T^{1} C^{18} G^{2}

C^{2} G^{17} A^{1} G^{3} C^{1} A^{1} G^{21} A^{1} T^{1} G^{1} A^{1} T^{2} G^{1} A^{6} G^{1} A^{37} G^{28} A^{1} G^{2} C^{1}

G^{1} T^{2} A^{1} T^{1} C^{8} T^{1} C^{15} A^{1} C^{22} A^{1} G^{2} T^{1} G^{1} C^{1} G^{6} C^{1} G^{35} A^{1} T^{14} C^{1}

T^{3} A^{1} T^{14} A^{1} T^{11} G^{1} C^{1} A^{2} T^{1} G^{1} T^{2} G^{1} T^{2} A^{1} G^{1} A^{7} T^{1} A^{22} T^{1} A^{5}

C^{1} A^{7} T^{4} A^{1} G^{1} T^{2} G^{1} T^{12} G^{1} T^{23} A^{1} T^{6} C^{1} T^{1} G^{1} T^{1} C^{1} T^{13} C^{1} T^{16}

A^{1} T^{13} G^{1} T^{2} G^{1} T^{1} A^{1} C^{1} G^{13} A^{3} G^{20} A^{1} G^{10} C^{1} T^{1} A^{1} G^{3} A^{1} G^{4} A^{1}

G^{1} A^{1} G^{5} A^{1} G^{1} C^{1} A^{1} G^{7} A^{1} G^{2} A^{2} G^{2} A^{5} G^{2} A^{2} T^{1} A^{2} T^{1} G^{1} A^{1}

C^{1} G^{3} C^{1} A^{3} C^{2} T^{2} C^{42} G^{1} C^{2} T^{1} A^{1} C^{1} G^{1} A^{2} C^{1} T^{19} C^{1} T^{47} G^{1} T^{21}

C^{1} T^{2} A^{2} T^{1} C^{3} T^{1} A^{2} C^{1} A^{9} T^{1} A^{8} T^{1} A^{20} C^{1} T^{28} C^{1} T^{12} A^{1} T^{1} C^{1}

T^{1} C^{2} A^{1} C^{20} T^{1} C^{20} T^{2} C^{1} G^{1}

Figure 5 The RLBWT of the concatenation of the toy genomes shown in Figure 3, consisting of 449 runs (20 runs per line).

3,	2,	86,	88^{12},	41,	88^{6},	89,	41,	88,	87,	88^{4},	16,	63,	88^{11},	19,
55^{5},	79,	55^{25},	51,	55,	45,	55^{2},	58,	55,	64,	19,	6,	73,	79,	55,
79^{2},	45,	79^{2},	65,	79^{9},	45,	79^{5},	18,	79,	82,	12^{3},	70,	73,	6^{2},	67,
90,	6^{3},	10,	6^{3},	5,	6,	84,	73,	6,	73^{7},	87,	70,	30,	73^{2},	68,
59,	30,	73,	33,	73^{2},	60,	73^{3},	76,	73,	85,	73,	13,	73^{3},	4,	6^{3},
83,	6^{8},	4,	6^{8},	77,	73,	55,	19,	57,	19,	50,	19^{4},	50,	19,	50,
19^{3},	57,	19,	50,	19,	81,	50,	19^{6},	66,	19,	50,	19,	50,	19,	50^{3},
74,	78,	66,	50,	49,	12,	0,	40,	48,	73,	26,	34,	62,	7,	1,
22,	18,	22,	19,	50^{10},	8,	22^{12},	50,	22^{3},	50,	28,	50,	22^{17},	71,	22,
71,	22^{8},	72,	22^{11},	29,	44,	22^{21},	45,	55,	45,	27,	36,	17,	35,	22,
20,	23,	12,	47,	12^{9},	56,	12^{2},	80,	12^{2},	46,	12^{10},	61,	46,	12^{5},	79,
50,	64,	88,	32,	55,	11,	69,	38,	32^{2},	31,	32,	55,	32^{3},	52,	25,
45,	32,	37,	42,	32,	58,	32,	39,	32^{5},	53,	32,	75,	32^{3},	19,	32,
41,	43,	58,	45,	55,	9,	55^{14},	45,	55^{3},	45,	55,	54,	6,	22,	51,
55,	64,	19,	64,	78,	64^{7},	21^{2},	64^{6},	14,	64^{11},	21,	64,	24,	64^{5},	15,
64														

Figure 6 The RLBWT of the sequence shown in Figure 6, consisting of 226 runs (15 runs per line).

dictionary — all those starting with the suffix of the pattern we have processed so far — but it is possible to start a backward search in the index for the parse with a lexicographic range of phrases rather than with a single phrase.

When we reach the left end of the leftmost trigger string in the pattern, we can use the same bitvector to map back into the index for the text and match the remaining prefix of the pattern with that. While matching the pattern phrase by phrase against the index for the parse, we can either compare against phrases in the stored dictionary or just use Karp-Rabin hashes (allowing some probability of false-positive matches). We still have to parse the pattern, but that requires a single sequential pass, while FM-indexes in particular are known for poor memory locality. They key idea is that, ideally, we match most of the pattern phrase by phrase instead of character by character, reducing the number of cache misses.

We plan to reimplement two-level indexes for collections of similar genomes with RLFM-indexes for the collections themselves and CSAs, standard RLCSAs and our sped-up RLCSAs for the parses from Theorem 7 of those collections, and compare them experimentally. We also plan to try indexing minimizer digests with CSAs and RLCSAs.

5 Boyer-Moore-Li with two-level indexing

Olbrich, Büchler and Ohlebusch [22] recently showed how working with `rsync`-like parses of genomes instead of the genomes themselves can speed up multiple alignment. More specfically, they find and use as anchors finding maximal substrings (call multi-MUMs) of the parses that occur exactly once in each parse. In this section we speculate about how two-level indexing may similarly speed up searches for long maximal exact matches (MEMs). A MEM of a pattern $P[0..m-1]$ with respect to a text T is a substring $P[i..j]$ of P such that

- $P[i..j]$ occurs in T,
- $i = 0$ or $P[i-1..j]$ does not occur in T,
- $j = m-1$ or $P[i..j+1]$ does not occur in T.

Finding long MEMs is an important task in bioinformatics and there are many tools for it.

Li [14] gave a practical algorithm, called forward-backward, for finding all the MEMs of P with respect to T using FM- or RLFM-indexes for T and its reverse T^{rev}. Assume all the distinct characters in P occur in T; otherwise, we split P into maximal substrings consisting only of copies of characters occurring in T and find the MEMs of those with respect to T. We first use the index for T^{rev} to find the longest prefix $P[0..e_1]$ of P that occurs in T, which is the leftmost MEM. If $e_1 = m - 1$ then we are done; otherwise, $P[e_1 + 1]$ is in the next MEM, so we use the index for T to find the longest suffix $P[s_2..e_1 + 1]$ of $P[0..e_1 + 1]$ that occurs in T. The next MEM starts at s_2, so conceptually we recurse on $P[s_2..m - 1]$. The total number of backward steps in the two indexes is proportional to the total length of all the MEMs.

Gagie [9] proposed a heuristic for speeding up forward-backward when we are interested only in MEMs of length at least L. We call this heuristic Boyer-Moore-Li, following a suggestion from Finlay Maguire [16]. Since any MEM of length at least L starting in $P[0..L-1]$ includes $P[L-1]$, we first use the index for T to find the longest suffix $P[s..L-1]$ of $P[0..L - 1]$ that occurs in T. If $s = 0$ then we fall back on forward-backward to find the leftmost MEM and the starting position of the next MEM. Otherwise, since we know there are no MEMs of length at least L starting in $P[0..s - 1]$, conceptually we recurse on $P[s..m-1]$. Li [15] tested Boyer-Moore-Li and found it practical enough that he incorporated it into his tool `ropebwt3`.

Suppose we build an `rsync`-like parse of $T[0..n - 1]$ and two-level indexes for T and T^{rev} based on that parse and parse P when we get it. With a naïve two-level version of Boyer-Moore-Li, we would simply use the two-level indexes in place of the normal FM- or RLFM-indexes for T and T^{rev}. We conjecture, however, that we can do better in practice.

Let $P[k]$ be the last character of the last phrase that ends strictly before $P[L]$, let $P[j]$ be the first character of the first phrase such that $P[j..k]$ occurs in T, and let $P[i]$ be the second character of the phrase preceding the one containing $P[j]$. Notice we can find i, j and k by matching phrase by phrase using only the top level (for the parse) of the two-level index for T. If $i > 0$ then we can immediately discard $P[0..i - 1]$ and conceptually recurse on $P[i..m - 1]$; otherwise, we proceed normally.

Of course, the value i is at most the value s found by regular Boyer-Moore-Li and could be much smaller, in which case discarding $P[0..i - 1]$ benefits us much less than discarding $P[0..s-1]$. We hope this is usually not the case and we look forward to testing Boyer-Moore-Li with two-level indexing.

References

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

2 Lorraine AK Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P Pissis. U-index: A universal indexing framework for matching long patterns. *arXiv:2502.14488*, 2025.

3 Paul Beame and Faith E Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65:38–72, 2002.

4 Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *Proceedings of the 20th International Symposium on Experimental Algorithms (SEA)*, 2022.

5 Francisco Claude, Patrick K Nicholson, and Diego Seco. On the compression of search trees. *Information Processing & Management*, 50:272–283, 2014.

6 Jin-Jie Deng, Wing-Kai Hon, Dominik Köppl, and Kunihiko Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the Data Compression Conference (DCC)*, 2022.

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

8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52:552–581, 2005.

9 Travis Gagie. How to find long maximal exact matches and ignore short ones. In *Proceedings of the 28th Conference on Developments in Language Theory (DLT)*, 2024.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67:1–54, 2020.

11 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35:378–407, 2005.

12 Aaron Hong, Marco Oliva, Dominik Köppl, Hideo Bannai, Christina Boucher, and Travis Gagie. PFP-FM: an accelerated FM-index. *Algorithms for Molecular Biology*, 19:15, 2024.

13 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Symposium on Combinatorial Pattern Matching (CPM)*, 2009.

14 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28:1838–1844, 2012.

15 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40:btae717, 2024.

16 Finlay Maguire. Personal communication, 2024.

17 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12:40–66, 2005.

18 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17:281–308, 2010.

19 Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3:25–47, 2000.

20 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2021.

21 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

22 Jannik Olbrich, Thomas Büchler, and Enno Ohlebusch. Generating multiple alignments on a pangenomic scale. *Bioinformatics*, 41(3):btaf104, 2025.

23 Alberto Ordóñez, Gonzalo Navarro, and Nieves R Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017.

24 Jouni Sirén. *Compressed Full-Text Indexes for Highly Repetitive Collections*. PhD thesis, University of Helsinki, 2012.

25 Jukka Teuhola. Interpolative coding of integer sequences supporting log-time random access. *Information Processing & Management*, 47:742–761, 2011.

26 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27, 2024.

27 Alan Zheng, Ishmeal Lee, Vikram S. Shivakumar, Omar Y. Ahmed, and Ben Langmead. Fast and flexible minimizer digestion with digest. *bioRxiv*, 2025. URL: `https://www.biorxiv.org/content/early/2025/01/08/2025.01.02.631161`.