Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections

Jouni Sirén[1⋆], Niko Välimäki[1⋆⋆], Veli Mäkinen[1⋆⋆], and Gonzalo Navarro[2⋆⋆⋆]

[1] Dept. of Computer Science, Univ. of Helsinki, Finland.
{jltsiren,nvalimak,vmakinen}@cs.helsinki.fi
[2] Dept. of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. A repetitive sequence collection is one where portions of a *base sequence* of length n are repeated many times with small variations, forming a collection of total length N. Examples of such collections are version control data and genome sequences of individuals, where the differences can be expressed by lists of basic edit operations. This paper is devoted to studying ways to store massive sets of highly repetitive sequence collections in space-efficient manner so that retrieval of the content as well as queries on the content of the sequences can be provided time-efficiently. We show that the state-of-the-art entropy-bound full-text *self-indexes* do not yet provide satisfactory space bounds for this specific task. We engineer some new structures that use run-length encoding and give empirical evidence that these structures are superior to the current structures.

1 Introduction

Self-indexing [9, 5, 24, 20] is a new algorithmic approach to storing and retrieving sequential data. The idea is to represent the text (a.k.a. sequence or string) compressed so that random access to the content of the text is maintained, and pattern retrieval queries on the content of the text are supported as well.

The self-indexing approach becomes especially interesting when applied to collections of texts. A special case of a text collection is one which contains several *versions* of one or more *base sequences*. Such collections are not uncommon. For example, a *version control system* needs to store several versions of the same file with only small edit differences between the consecutive entries. If the entries are stored independently of each others, the version control system will end up spending unnecessarily large amounts of memory. If the system stores only the edits, queries on the content of one specific version becomes non-trivial.

An analogy to the storage and retrieval of version control data is soon becoming reality in the field of molecular biology. Once the DNA sequencing technologies become faster and more cost-effective, it may be that in the near future the

⋆ Funded by the Research Foundation of the University of Helsinki.
⋆⋆ Funded by the Academy of Finland under grant 119815.
⋆⋆⋆ Partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

sequencing of individual genomes becomes a feasible task [3, 12, 21]. With such data in hand, many fundamental issues such as storing and analyzing thousands of individual genomes become a top concern. For the analysis of such collections of biological sequences, one would need to use some variant of a *generalized suffix tree* [11] as that provides a variety of algorithmic tools to do analysis in linear or near-linear time. The memory requirement of such solution is unimaginable with current random access memories and also challenging in permanent storage.

Self-indexes should, in principle, cope well with the two applications above as both data types contain high amounts of repetitive structure. In particular, as the main building blocks of *compressed suffix trees* [25, 23, 22, 7] they enable compressing the collections to close to their *high-order entropy* and enabling flexible analysis tasks to be executed. However, there is a fundamental problem with the fact that the high-order entropies are defined by the frequencies of symbols in their fixed-length contexts. These contexts do not change *at all* when more *identical* sequences are added to the collection. Hence, these self-indexes are unable of exploiting the fact that the texts in the collection are highly similar.

In this paper, we propose new self-indexes based on run-length compression, that are suitable for storing highly repetitive collections of texts. We implemented the new structures and compared them experimentally to existing structures. The experiments show that our new structures achieve superior compression both on DNA collections and on version control data. The superiority can be explained in theory as well; the theoretical analysis together with related extended results (see Sect. 7) is part of subsequent work [16].

The paper is structured as follows. Section 2 introduces the basic concepts and goes through the related literature. Sections 3, 4, and 5 derive the new run-length compressed indexes. Section 6 gives the experimental results and Sect. 7 discusses the subsequent work.

2 Basic Concepts

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a sequence of *symbols* (a.k.a. character or letter). Each symbol is an element of a *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written $S_{i,j} = s_i s_{i+1} \ldots s_j$. A *prefix* of S is a substring of the form $S_{1,j}$, and a *suffix* is a substring of the form $S_{i,n}$. If $i > j$ then $S_{i,j} = \varepsilon$, the empty string of length $|\varepsilon| = 0$. A *text* string $T = T_{1,n}$ is a special string with $t_n = \$$. The *lexicographical order* "$<$" among strings is defined in the obvious way.

We assume the reader is familiar with the *empirical k-th order entropy* $H_k(T)$ for which holds $0 \leq H_k(T) \leq H_{k-1}(T) \leq \cdots \leq H_0(T) \leq \log \sigma$ [18].

The compressors to be discussed are derivatives of the *Burrows-Wheeler transform (BWT)* [2]. The transform produces a permutation of T, denoted by T^{bwt}, as follows: (i) Build *suffix array* [17] $\mathsf{SA}[1, n]$ of T, that is an array of pointers to all the suffixes of T in the lexicographic order; (ii) The transformed text is $T^{bwt} = L$, where $L[i] = T[\mathsf{SA}[i] - 1]$, taking $T[0] = T[n]$.

The BWT is reversible, that is, given $T^{bwt} = L$ we can obtain T as follows: (a) Compute the array $C[1, \sigma]$ storing in $C[c]$ the number of occurrences of

characters $\{\$, 1, \ldots, c-1\}$ in the text T; (b) Define the *LF mapping* as follows: $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in the prefix $L[1, i]$; (c) Reconstruct T backwards as follows: set $s = 1$, for each $n-1, \ldots, 1$ do $t_i \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally, append the end marker $t_n \leftarrow \$$. We study the following problem.

Definition 1. *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^1| = n$ and $\sum_{k=1}^{r} |T^k| = N$, where $T^2, T^3, \ldots, T^r$ contain overall s mutations (i.e., symbol substitutions) from the base sequence T^1, the* repetitive collection indexing problem *is to store $\mathcal{C}$ in as small space as possible such that the following operations are supported as efficiently as possible:* `count`(P) *(How many times P appears as a substring of the texts in $\mathcal{C}$?);* `locate`(P) *(List the occurrence positions of P in $\mathcal{C}$); and* `display`(k, i, j) *(Return $T^k_{i,j}$).*

The above is an extension of the well-known *basic indexing problem*, where the collection has only one sequence T. We call a data structure a *self-index* if it does not need T to solve the three queries above.

A comprehensive solution to the basic indexing problem uses the suffix array $\mathsf{SA}[1, n]$. Two binary searches are enough to find the interval $\mathsf{SA}[sp, ep]$ such that `count` and `locate` are immediately solved [17]. The solution is not as space-efficient as possible, since array SA requires $n \log n$ bits, and the solution is not yet a self-index, since T is needed.

The *FM-index* [5] is a self-index based on the BWT. It solves counting queries by finding the interval $\mathsf{SA}[sp, ep]$ that contains the occurrences of pattern P. The FM-index uses the array C and function $rank_c(L, i)$ in the so-called *backward search* algorithm, calling function $rank_c(L, i)$ $O(m)$ times. The two other basic indexing problem queries are solved e.g. using sampling of SA and its inverse SA^{-1}, and LF-mapping to derive the unsampled values from the sampled ones. Many variants of the FM-index have been derived that differ mainly in the way the $rank_c(L, i)$-queries are solved [20]. For example, on small alphabet sizes, one can achieve $nH_k(1 + o(1))$ space with constant time support for $rank_c(L, i)$ [6].

Now, the (repetitive) collection indexing problem can be solved using the normal self-index for the concatenation $T^1 \# T^2 \# \cdots T^r\$$, where $\# \notin \Sigma$ is a special symbol. However, the space requirement achieved even with a high-entropy compressed index is not attractive for the case of repetitive collections. For example, the solution by Ferragina et al. [6] requires $NH_k(C) + o(N \log \sigma)$ bits. Notice that even with $s = 0$, $H_k(C) \approx H_k(T^1)$, and hence the space is about r times more than what the same solution uses for the basic indexing problem.

In this paper, we derive solutions whose space requirements depend on the number of *runs in the Burrows-Wheeler transform*. We will introduce some notations to talk about runs. A *self-repetition* is a maximal interval $\mathsf{SA}[i, i+l]$ of suffix array SA having a *target interval* $\mathsf{SA}[j, j+l]$ such that $\mathsf{SA}[j+r] = \mathsf{SA}[i+r]+1$ for all $0 \leq r \leq l$. Let $\Psi(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]+1]$ [9, 24]. The intervals of Ψ corresponding to a self-repetition in the suffix array are called *runs*. We have $\Psi(i+1) = \Psi(i)+1$ when both $\Psi(i)$ and $\Psi(i+1)$ are contained in the same run.

Let $R_\Psi(T)$ be the number of runs in Ψ of text T and $R(T) = R_{bwt}(T)$ the number of equal letter runs in T_{bwt}. Both are tightly connected, R_Ψ and R_{bwt},

namely $R_\Psi \leq R_{bwt} \leq R_\Psi + \sigma$ [14], allowing one to use them interchangeably under most circumstances. We will denote both with R when clear from context.

Now, it is easy to see that quantities $R_{bwt}(T)$ and $R_{bwt}(\mathcal{C})$ are the same when $s = 0$. Mutations make $R_{bwt}(\mathcal{C})$ grow. It is possible to derive expected case bounds on how these terms are related; these analyses are omitted here. Instead, we introduce structures whose space depends on $R_{bwt}(\mathcal{C})$ and study empirically the growth of $R_{bwt}(\mathcal{C})$ on varying s. We limit our attention to self-indexes providing query $\texttt{count}(P)$.

3 RLCSA: Run-Length Compressed Suffix Array

The *Run-Length Compressed Suffix Array* is based on the Compressed Suffix Array by Mäkinen, Navarro and Sadakane [15]. We use run-length encoding of the differences $\Psi(i) - \Psi(i-1)$ to store the array. Absolute $\Psi(i)$ values are sampled at regular intervals of the *compressed* array. The resulting structure supports counting queries with backward searching.

Differential encoding of Ψ transforms a run $\Psi(i)\Psi(i+1)\cdots\Psi(i+l)$ into $\Psi(i) - \Psi(i-1)$ followed by l 1s, where $\Psi(i) - \Psi(i-1) > 1$. We say that the run is *trivial* if $l = 0$. If we use run-length encoding on the 1s, we encode the trivial runs simply as $\Psi(i) - \Psi(i-1)$. A nontrivial run, instead, is encoded as three numbers, $\Psi(i) - \Psi(i-1), 1, l$. That is, each time we encode a difference equal to 1, the length of the run of 1s follows. This way, run-length compression pays nothing for trivial runs, only for nontrivial runs where it has a potential benefit.

Let N be the total size of the collection and R' the number of nontrivial runs. The sum of all the differences $\Psi(i) - \Psi(i-1)$ is at most σN [15], and the total length of the runs of 1s is $N - R$. Hence by using Elias delta coding to encode the integers, we need at most

$$|\Psi| \leq \left(R \log \frac{\sigma N}{R} + R' \left(1 + \log \frac{N-R}{R'} \right) \right) (1 + o(1))$$

bits for the array Ψ. By using sampling step of B bits, we need $O((\frac{|\Psi|}{B} + \sigma) \log N)$ bits for the sampled $\Psi(i)$ values, effectively making the total size of RLCSA $|\Psi|(1+\varepsilon)$ for any $\varepsilon > 0$.

To retrieve $\Psi(i)$, we first binary search the samples and then sum up the differences in the corresponding part of the Ψ array until we reach position i. This gives us $\texttt{count}(P)$ queries in $O(|P|(\log \frac{|\Psi|}{B} + B))$ time by using bacward searching [15].

4 RLWT: Run-Length Encoded Wavelet Tree

Next we will describe a new data structure that we call *Run-Length encoded Wavelet Tree*. We exploit well-known bit-vector operations: For a bit vector B of length u, $rank_b(B, i)$ gives the number of b-bits in $B[1, i]$ for all $1 \leq i \leq u$

and $b \in \{0,1\}$. The inverse function $select_b(B,x)$ gives the position of the x'th b-bit in the bit vector B.

Wavelet tree [8] is a binary tree structure whose leaves represent the symbols in the alphabet. The root is associated with the sequence $T = T_{1,N}$. In a *balanced* wavelet tree, the left (right) child of the root is a wavelet tree of the sequence $T_<$ ($T_\geq$) obtained by concatenating all positions i having $t_i < \sigma/2$ ($t_i \geq \sigma/2$). This subdivision is represented by a bit vector of length n that marks which positions go to the left subtree (by 0) and which go right (by 1). Recursion is continued until the concatenated sequence contains a repeat of one symbol. One can reveal t_i, compute $rank_c(T,i)$, and $select_c(T,j)$ with $O(\log\sigma)$ *rank*/*select* queries on the bit-vectors on the path to the leaf (or back) containing c [8].

The space required by a balanced wavelet tree depends on how we encode the bit vectors. Let R be the number of runs in a text $T_{1,N}$. Let B^{all} be the level-wise concatenation of all the bit vectors in the balanced wavelet tree for the sequence T. In the worst case, each run in T equals one 0/1-bit run on each of the $\log\sigma$ levels of the wavelet tree, so that the upper-bound for the number of 0/1-bit runs in B^{all} is $R\log\sigma$ (the best case is $1\cdot\log\sigma$). Let $b \leq \lceil \frac{1}{2}R\log\sigma \rceil$ be the number of 1-bit runs in B^{all}. The RLWT data structure encodes B^{all} into two separate bit vectors B^1 and B^{rl} such that the number of 1-bits in both bit vectors is exactly b: bit vector B^1 marks all the starting positions of 1-bit runs in B^{all}, and bit vector B^{rl} encodes the run-lengths of these runs in *unary coding*. More precisely, $B^1[i] = 1$ only if $B^{all}[i] = 1$ and $B^{all}[i-1] = 0$, for all $1 < i \leq N\log\sigma$, and $B^1[1] = 1$ if $B^{all}[1] = 1$. Unary code for a bit run of length j contains $j-1$ zero bits concatenated with one 1-bit. The length of B^{rl} is the sum of the lengths of 1-bit runs in B^{all}, which is always at most $N\log\sigma$ bits.

Query $rank_1(B^{all},i)$ can be solved using only the bit vectors B^1 and B^{rl} by calculating the number of 1-bits in two closed intervals $[0,j-1]$ and $[j,i]$, where j is the starting position of the 1-bit run that precedes position i in B^{all}. For the first interval, let r be the number of 1-bit runs in B^{all} that start before or at the position i, i.e. $r = rank_1(B^1,i)$. From the definition of B^{rl} follows that $rank_1(B^{all},j-1)$ equals $select_1(B^{rl},r-1)$. Now it remains to calculate the number of 1-bits in the closed interval $[j,i]$ of the bit vector B^{all}: Let k be the length of the rth run, that is to say $k \leftarrow select_1(B^{rl},r) - rank_1(B^{all},j-1)$. The number of 1-bits in the closed interval is

$$rank_1(B^{all},i) - rank_1(B^{all},j-1) = \begin{cases} k & \text{if } i-j \geq k, \\ i-j+1 & \text{otherwise.} \end{cases}$$

Finally, the answer to the original $rank_1(B^{all},i)$ query is just the sum of the above values $rank_1(B^{all},j-1)$ and $rank_1(B^{all},i) - rank_1(B^{all},j-1)$.

Gupta et al. [10] have shown that a *binary searchable dictionary representation (BSD)* of a bit-vector B of u bits containing b 1-bits, requires $|gap(B)| + O(|gap(B)|/\log b) = |gap(B)|(1+o(1))$ bits of space and supports *rank* queries in $t_{\mathsf{AT}} = AT(u,b)$ time, where $AT(u,b) = o((\log\log u)^2)$, and and *select* in $O(\log\log b)$ time. In the worst case, length of the *gap encoded sequence* $|gap(B)|$ is $b\log(u/b) + O(b\log\log(u/b))$ bits.

For the bit vectors B^1 and B^{rl}, we have strict upper-bounds of $u \leq N \log \sigma$ and $b \leq \lceil \frac{1}{2} R \log \sigma \rceil$. Using the BSD, the bit vectors can be represented in at most $R \log \sigma \log \frac{2N}{R}(1 + o(1)) + O\left(R \log \sigma \log \log \frac{2N}{R}\right)$ bits. All the wavelet tree queries can be supported without storing the bit vector B^{all} itself.

Using the RLWT structure with *backward searching* [5], we can count the number of occurrences of a pattern of length m in $O(m \log(\sigma t_{\mathsf{AT}}))$ time. Table C adds $\sigma \log N$ bits to the space requirement.

5 RLFM+: Improved Run-Length FM-index

The RLWT structure can be improved in the case the input text is T^{bwt}: The *Run-Length FM-Index (RLFM)* of [14] uses a reduction such that the equal letter runs of T^{bwt} are marked into two bit-vectors, and the sequence of run heads of length R is encoded using a normal wavelet tree. We can represent the two bit-vectors using BSD, giving immediately the following result: The RLFM data structure for the sequence T^{bwt} takes $(R \log \sigma + 2R \log \frac{N}{R})(1+o(1)) + O\left(R \log \log \frac{N}{R}\right) + \sigma \log N$ bits of space.The structure supports `count`(P) in time $O(|P|(\log(\sigma) + t_{\mathsf{AT}}))$.

6 Experimental Results

We implemented the three proposed structures RLCSA, RLWT, and RLFM+, each supporting `count`()-queries. Standard strategies to support `display`() and `locate`() are trivial to add. (Almost all space/time tradeoffs are possible for those queries, so the base structure for supporting `count`() is the crucial one.)

For comparison, we selected several well-engineered implementations of self-indexes from the *Pizza&Chili* site[3]. Unless otherwise noted, we used no extra space for `display()` and `locate()`, and left the default options for the rest. We also compared our indexes to several compressors and a version control system.

We performed experiments on two data sets. The synthetic DNA sequence collections were based on the DNA sequences from *Pizza&Chili*. We took a 1, 4 or 16 MB prefix and repeated it 25, 50 or 100 times. Each character in the repetitions was individually mutated into another character in $\{A, C, G, T\}$ with ten different probabilities ranging from 0 to 0.05. This was intended to simulate the case of one base sequence and $r - 1$ mutated sequences.

Our other data set is based on the source code for portable versions of OpenSSH[4]. We used a 4.44 MB `tar` archive containing the source code for version 4.7p1, as well as on another 176.55 MB archive containing the source code for all 75 versions up to version 4.7p1. The latter contained multiple copies of the same files as well as many highly similar files, making it highly compressible.

The experiments were performed on a 3 GHz Intel Pentium 4 Northwood machine with 3 GB RAM running Fedora Core 7 based Linux.

[3] `http://pizzachili.dcc.uchile.cl/` or `http://pizzachili.di.unipi.it/`.
[4] `http://www.openssh.com/`

6.1 Implementations and Parameters

The implementations of Succinct Suffix Array (SSA, version 2) [14, 6], Run-Length FM-index (RLFM) [14], Alphabet-Friendly FM-index (AFFM, version 2) [6] were taken from the *Pizza&Chili* site. All of them use a Huffman-shaped wavelet tree to achieve compression. SSA achieves zero-order compression by building the wavelet tree directly on the BWT, and is the fastest. RLFM builds it on the run heads of the BWT, and thus its space is related to the number of runs in the collection, yet the two extra bit-vectors it uses are not compressed. AFFM achieves high-order compression, $NH_k + o(N \log \sigma)$ bits, by partitioning the BWT into suitable chunks and building a wavelet tree per chunk. Its space is not related to the runs in the BWT.

Sadakane's Compressed Suffix Array (CSA) [24] implementation was also taken from *Pizza&Chili*. It achieves high-order compression related to the runs in Ψ, yet also includes less compressed bit vectors. We used sample rates 128 (default; CSA-128) and 1024 (CSA-1024 or CSA) for the Ψ values. The total size of the samples for a 400 MB collection is 3.1 MB for CSA-1024 and 25 MB for CSA-128. Suffix array sample rate was set to 65536 to make the size of these unused samples negligible (not to confuse with the sampling to access Ψ).

Also included in the comparison was a self-index based on Lempel-Ziv parsing (LZ-index, *Pizza&Chili* version 4) [1]. We selected $1/\epsilon = 15$ as a reasonable space/time tradeoff and subtracted the space (41 MB for a 400 MB collection) used for `display()` and `locate()` queries, for fairness with the other structures (although the implementation does not let one discard it).

Our indexes RLCSA, RLWT, and RLFM+ can be seen as versions of CSA, SSA, and RLFM, respectively, enhanced to profit from highly repetitive collections. The implementation of RLCSA is optimized for secondary memory. Hence we have used 32 kilobyte sampling step for Ψ (RLCSA-32k or RLCSA) in addition to the more reasonable 128 bytes (RLCSA-128). In practice, RLCSA-128 is at most 20% larger than RLCSA-32k. The difference can be reduced by changing the size of the samples from 24 bytes to $3\lceil \log N \rceil$ bits per sample. In RLWT and RLFM+, we used simpler encoding for the bit vectors than the original BSD. The implemented structure solves *rank* in $O(\log b)$ time.

In addition to the existing self-indexes, we compared our new indexes to several plain compressors. The well-known `gzip` and `bzip2` compressors were used with parameter `-9` to achieve maximum compression. Due to their small block sizes, they cannot profit from the large-scale repetitiveness in our data sets. We have also used the highly efficient LZ77-based compressor `p7zip`[5] with options `-mx=9 -md=30` to see how much we pay for the retrieval functionality. With a window of length up to 1 GB, `p7zip` can compress texts with long repeats much better than standard Lempel-Ziv based compressors.

Finally we have used the Subversion (SVN)[6] version control system for the OpenSSH source code data set. The source codes were inserted into a reposi-

[5] `http://p7zip.sourceforge.net/`
[6] `http://subversion.tigris.org/`

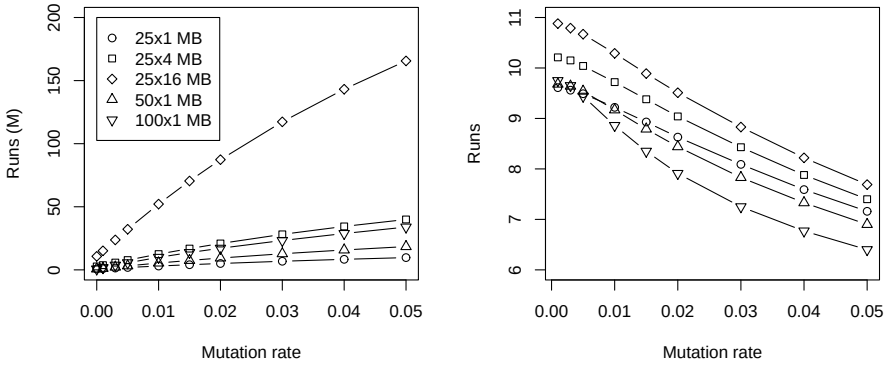

Fig. 1. The number of runs in Ψ (left) and the average number of new runs per mutation (right) on repeated DNA sequences.

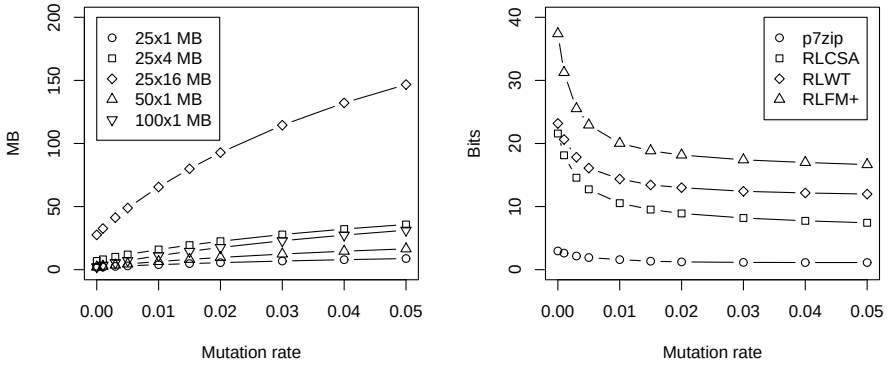

Fig. 2. The size of RLCSA on repeated DNA sequences (left) and the average number of bits required to encode a run on 25x16 MB DNA (right).

tory using FSFS file system in a chronological order one version at a time. We measured the sizes of subdirectory `db/` of the repository, using utility `du`.

6.2 Results

Fig. 1 shows the number of runs in Ψ of repeated DNA sequences. The number of runs grows somewhat sublinearly in the number of mutations. New runs are created when mutations move suffixes to new positions in the suffix array. However, as the mutations accumulate, it becomes more likely that a similar mutation has already happened before, reducing the number of new runs created.

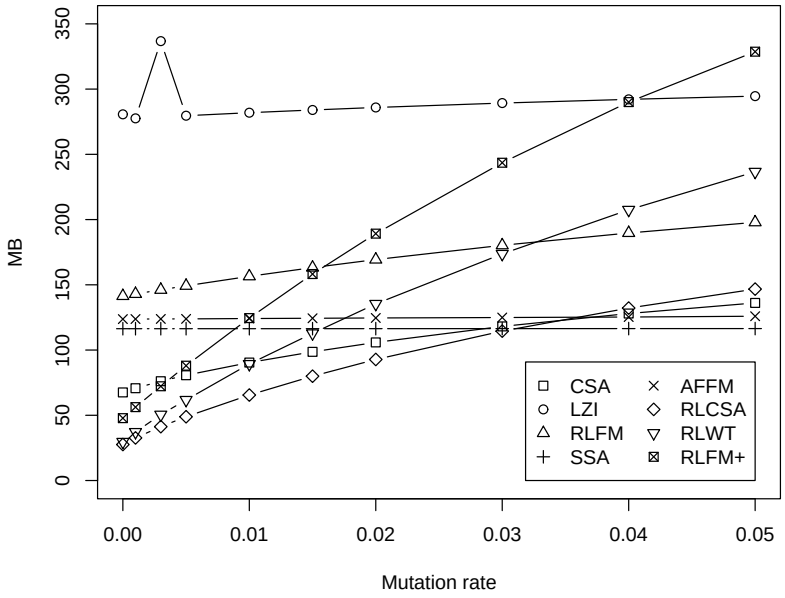

Fig. 3. A comparison of our indexes with existing self-indexes. The peak of LZI at 0.003 is an artifact of the implementation.

Fig. 2 shows the sensitivity of the sizes of our new self-indexes to the number of runs in the collection. RLCSA is clearly smaller than the two other indexes. It is interesting to see that with high mutation rates, `p7zip` requires only about 1.2 bits per run, suggesting some connection between the number of runs in Ψ and the space requirements of Lempel-Ziv compression (see Sect. 7).

We select the 25 times repeated 16 MB DNA prefix for comparisons between new and existing self-indexes. As Fig. 3 shows, our indexes clearly outperform the existing self-indexes when the number of mutations is small. In particular, it can be seen that our indexes are the most sensitive to high repetitiveness, followed by CSA and RLFM, and then LZ-index. SSA and AFFM are completely insensitive.

As predicted by the theoretical space bounds, RLCSA outperforms RLWT. Surprisingly, RLWT outperforms RLFM+. This is explained by the fact that RLFM+ *always* uses two bit-vectors with R bits set, and a separate wavelet tree taking close to $R \log \sigma$ bits (or slightly less in practice due to the Huffman shape). RLWT instead uses a wavelet tree formed by $\log \sigma$ levels of bit vectors each with *at most* R bits set. This worst case does not happen in practice. On random text the expected number of bits set is $\frac{\sigma/2}{\sigma-1} R \log \sigma$, and this decreases on non-random text due to the BWT effect. For example on DNA ($\log \sigma = 2$) there are only $1.25R$ bits set in RLWT. Assuming a δ-encoding of the run lengths, we get a pretty good approximation of 14.34 bits for RLWT, and 18.58 for RLFM+.

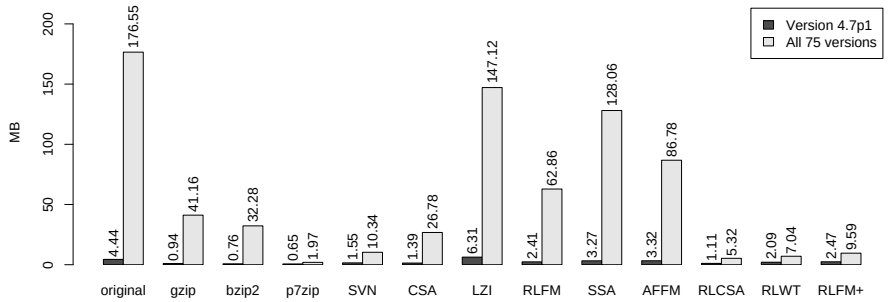

Fig. 4. Compression results for OpenSSH sources.

Table 1. Time for counting on the different indexes. We remind that the LZ-index is not designed for counting.

Structure	Time (μs)	Size (MB)
CSA-128	103.0	112.29
CSA-1024	347.0	90.41
LZ-index	198596.8	281.92
RLFM	29.5	156.50
SSA	13.0	116.37
AFFM	19.4	124.15

Structure	Time (μs)	Size (MB)
RLCSA-128	72.7	77.54
RLCSA-32k	11130.0	65.52
RLWT	1050.0	89.30
RLFM+	189.7	124.48

The size difference between RLCSA and RLFM+ is also surprising, given the similar high-order terms in the space bounds. This is partially explained by the ratio of non-trivial runs to total runs R'/R decreasing from 0.80 at mutation rate 0.001 to 0.37 at 0.05. Additionally, the size bound for RLFM+ has a significant low-order term. Also note the size difference of the similar CSA and RLFM.

Next we compare our indexes with existing self-indexes as well as plain compressors on OpenSSH sources. As seen in Fig. 4, our indexes clearly outperform the existing self-indexes. Again RLWT outperforms RLFM+ even with this larger alphabet size, indicating that the average RLWT space requirement is better than the worst case (see also [4] for a more rigorous analysis of runs in wavelet tree). It is interesting to note that despite the search functionality, our indexes remain smaller than the SVN repository.

The increased space efficiency of our indexes has been paid in time efficiency. To test this, we extract 1000 random substrings of length 10 from the 16 MB DNA prefix. We then repeat the prefix 25 times with mutation rate 0.01 and measure counting query times. Table 1 gives average query times and structure sizes, showing the competitiveness of RLCSA-128.

7 Discussion

In this study, we have mainly considered self-indexes based on the Burrows-Wheeler framework. There is also a family of (self-)indexes which is based on the Lempel-Ziv parsing, see [13, 19, 20]. It is easy to see that the LZ77 parsing of a repetitive text collection consists of at most of $P(T^1) + s + 1$ phrases, where $P(T^1)$ gives the number of phrases in T^1. It follows that LZ77 based indexes require at most $O(n \log \sigma + s \log n)$ bits of space. However, there does not exist a LZ77 based *self-index*, as they require the uncompressed text to operate. All the Lempel-Ziv self-indexes (like the one experimented here) are based on the LZ78 parsing, which does not guarantee equally good performance. Hence, a promising future direction is to develop LZ77 based self-indexes.

Our experiments considered only point mutations on DNA, although there are many other types of mutations, like insertions, deletions, translocations, and reversals. The runs in the Burrows-Wheeler transform change only for those suffixes whose lexicographic order is affected by a mutation. In all mutation types (except in reversals[7]) the effect is identical to point mutations, so the compression result should be similar. We emphasize that the proposed indexes are completely universal, as they do not need to know what and where the mutations are. This is also illustrated by the experiment on version control data, where the changes are cumulative, and there is no base sequence, but rather a "founder sequence". The founder model also characterizes genome collections, but again the index does not need to know the phylogeny to succeed in compression.

In subsequent (still theoretical) work [16], we have derived dynamic versions of all the proposed self-indexes, where sequences can be deleted from and inserted to the collection at any time. These indexes take basically the same space as the static ones discussed here. We have also considered new structures for `display` and `locate`, where the number of suffix array samples depend on s as well. One can use both the static and the dynamic versions of these indexes as building blocks of recent compressed suffix trees [23, 22, 7].

References

1. D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th CPM*, LNCS 4009, pages 319–330, 2006.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.
3. G. M. Church. Genomes for all. *Scientific American*, 294(1):47–54, 2006.
4. P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. 33rd ICALP*, LNCS 4051, pages 560–571, 2006.
5. P. Ferragina and G. Manzini. Indexing compressed texts. *J. of the ACM*, 52(4):552–581, 2005.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.

[7] Adding the reverse complement of the base sequence to the collection solves this.

7. J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th CPM*, LNCS 5029, pages 152–165, 2008.
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
9. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. on Computing*, 35(2):378–407, 2006.
10. A. Gupta, W.-K. Hon, R. Shah, and J.S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. 16th DCC*, pages 213–222, 2006.
11. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
12. N. Hall. Advanced sequencing technologies and their wider impact in microbiology. *The Journal of Experimental Biology*, 209:1518–1525, 2007.
13. J. Kärkkäinen. Repetition-based text indexes. Technical Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.
14. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
15. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th ISAAC*, LNCS 3341, pages 681–692, 2004.
16. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Run-length compressed indexes for repetitive sequence collections. Technical Report C-2008-42, Department of Computer Science, University of Helsinki, Finland, 2008.
17. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.
18. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. of the ACM*, 48(3):407–430, 2001.
19. G. Navarro. Indexing text using the ziv-lempel trie. *J. of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
20. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
21. E. Pennisi. Breakthrough of the year: Human genetic variation. *Science*, 21:1842–1843, December 2007.
22. L. Russo, G. Navarro, and A. Oliveira. Dynamic fully-compressed suffix trees. In *Proc. 19th CPM*, LNCS 5029, pages 191–203, 2008.
23. L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th LATIN*, LNCS 4957, pages 362–373, 2008.
24. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.
25. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.