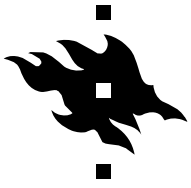
Department of Computer Science
Series of Publications C
Report C-2008-1

Storage and Retrieval of Individual Genomes and other Repetitive Sequence Collections

Veli Mäkinen, Jouni Sirén, and Niko Välimäki



University of Helsinki
Finland

Storage and Retrieval of Individual Genomes and other Repetitive Sequence Collections

Veli Mäkinen, Jouni Sirén, and Niko Välimäki

Department of Computer Science
P.O. Box 68, FIN-00014 University of Helsinki, Finland
{vmakinen,jltsiren,nvalimak}@cs.helsinki.fi

Technical report, Series of Publications C, Report C-2008-1
Helsinki, February 2008, 18 pages

Abstract

In the near future, biomolecular engineering techniques will reach a state where the sequencing of individual genomes becomes feasible. This progress will create huge expectations for the data analysis domain to reveal new knowledge on the "secrets of life". Quite rudimentary reasons may inhibit such breakthroughs; it may not be feasible to store all the data in a form that would enable anything but most basic data analysis routines to be executed. This paper is devoted into studying ways to store massive sets of complete individual genomes 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 although the state-of-the-art full-text *self-indexes* do not yet provide satisfactory space bounds for this specific task, after carefully engineering those structures it is possible to achieve very attractive results; the new structures are fully able to exploit the fact that the individual genomes are highly similar. We confirm the theoretical findings by experiments on large DNA sequences, and also on version control data, that forms another application domain for our methods.

Computing Reviews (1998) Categories and Subject Descriptors:
E.4 Coding and Information Theory — data compaction and compression
F.2.2 Analysis of Algorithms and Problem Complexity: Nonnumerical Algorithms and Problems — pattern matching, sorting and searching

General Terms:
Algorithms, Compression, Data structures

Additional Key Words and Phrases:
combinatorial pattern matching, data structure compression, full-text indexing

Contents

1 Introduction

Self-indexing [8, 5, 23, 18] is a new proposal for 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. Consider for example a file system that is automatically kept self-indexed. Files can be uncompressed when accessed and applying "find" command works in real-time as queries can be answered efficiently using the index rather than scanning through the files. Such retrieval functionalities have been available long time for natural language texts by the well-known *inverted indexes*, but now the self-indexes make such retrieval possible for arbitrary texts such as biological sequences that do not consist of separable words.

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 be spending unnecessarily large amounts of memory by time. 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 sequencing of individual genomes becomes a feasible task [3, 11, 20]. With such data in hand, many fundamental issues become of top concern, like how to store e.g. $10,000$ Human Genomes not to speak about analyzing them. For the analysis of such collections of biological sequences, one would clearly need to use some variant of a *generalized suffix tree* [10]as that provides a variety of algorithmic tools to do analyzes in 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* [24, 22] they enable compressing the collections in consideration 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 not at all able to exploit the fact that the texts in the collection are highly similar. Also, most self-indexes contain significant sub-linear terms that disappear very slowly with the collection size growth.

In this paper, we propose new self-indexes that are suitable for storing highly repetitive collections of texts. We analyze the theoretical space-requirements of these structures and show that they achieve much better bounds for this specific problem than the existing self-indexes. We implemented the structures and the experiments show that the theoretical advantages can also be seen in practice.

The paper is structured as follows. Section 2 introduces the basic concepts and goes through the related literature. Section 3 derives the bounds for the backbone of the two data structures that are presented in Sects. 4 and 5. Section 6 gives the experimental

results and Sect. 7 discusses the work left for future.

2 Basic Concepts and Background

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 finite *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 *subsequence* of S is any string obtained by deleting zero or more symbols from S.

A *text* string $T = T_{1,n}$ is a special string with $t_n = \$$. The *k-context* $C_i \in \Sigma^k$ of a symbol t_i is $C_i = t_{i+1 \bmod n} t_{i+2 \bmod n} \cdots t_{i+k \bmod n}$.[1] The *lexicographical order* "<" among strings is defined in the obvious way.

For discussing the compressibility of text collections we need the following concepts.

Definition 1 *The* zero-order empirical entropy *of text T is defined as*

$$H_0 \quad = \quad H_0(T) \quad = \quad \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c},$$

where n_c is the number of occurrences of symbol c in T.

Definition 2 *The* k-th order empirical entropy *of text T is defined as*

$$H_k \quad = \quad H_k(T) \quad = \quad \sum_{C \in \Sigma^k} \frac{|T_C|}{n} H_0(T_C), \tag{1}$$

where T_C is the subsequence of T formed by all the symbols whose k-context in T is C.

The k-th order entropy gives a lower bound for the compressibility of T with any compressor that can use only k-contexts to predict the preceding symbol.

Notice that $H_k(T) \leq H_{k-1}(T) \leq \cdots \leq H_0 \leq n \log \sigma$ bits, where the latter gives the lower-bound for representing any incompressible text.

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: (1) Form a *conceptual* matrix $\mathcal{M}$ whose rows are the *cyclic shifts* ($t_i t_{i+1} \cdots t_n t_1 t_2 \cdots t_{i-1}$ for $1 \leq i \leq n$) of text T, call F its first column and L its last column; (2) sort the rows of $\mathcal{M}$ in lexicographic order; (3) the transformed text is $T^{bwt} = L$.

The BWT is reversible, that is, given T^{bwt} we can obtain T as follows:

1. Compute the array $C[1, \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T.

2. 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 $L[1, i]$.

3. Reconstruct T backwards as follows: set $s = 1$ (since $\mathcal{M}[1] = \$T_{1,n-1}$) and, for each $n-1, \ldots, 1$ do $t_i \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put the end marker $t_n \leftarrow \$$.

We study a generalization of the following problem:

[1] For technical convenience, T is taken here as a circular string, and C_i is defined as the right context and not as the more natural left context. Alternative definitions can be handled with small modifications.

Definition 3 *The* basic indexing problem *is to store text T in as small space as possible, so that the following retrieval queries on any given* pattern *string $P = p_1 p_2 \cdots p_m$ can be solved as efficiently as possible:*

`count`(P)**:** *How many times P appears as a substring of T?*

`locate`(P)**:** *List the occurrence positions of P in T.*

`display`(i, j)**:** *Return $T_{i,j}$.*

We call a solution to the basic indexing problem a *self-index* if the index does not need T to solve the three queries above.

A comprehensive solution to the basic indexing problem uses the *suffix array* $\mathcal{A}[1, n]$, that is an array of pointers to all the suffixes of T in lexicographic order. Then two binary searches are enough to find the interval $\mathcal{A}[sp, ep]$ such that `count` and `locate` are immediately solved [15]. The solution is not as space-efficient as possible, since array $\mathcal{A}$ requires $n \log n$ bits, and the solution is not yet a self-index, since T is needed in order to solve the `display` query.

A more space-efficient solution to the basic indexing problem derives from the fact that suffix array $\mathcal{A}$ is essentially the matrix $\mathcal{M}$, as sorting the cyclic shifts of T is the same as sorting its suffixes given the end marker "\$": $\mathcal{A}[i] = j$ if and only if the ith row of $\mathcal{M}$ contains the string $t_j t_{j+1} \ldots t_{n-1} \$ t_1 \ldots t_{j-1}$.

The *FM-index* [5] is a self-index based on the Burrows-Wheeler transform. It solves counting queries by finding the interval $\mathcal{A}[sp, ep]$ 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 $\mathcal{A}$ and its inverse, 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 [18]. For example, on small alphabet sizes, it is possible to achieve $nH_k(1 + o(1))$ space with constant time support for $rank_c(L, i)$ [6].

A dual approach to solving the basic indexing problem uses the *compressed suffix array (CSA)* [23], that is a self-index based on an earlier succinct data structure [8]. In the CSA, the suffix array $\mathcal{A}[1, n]$ is represented by a sequence of numbers $\Psi(i)$, such that $\mathcal{A}[\Psi(i)] = \mathcal{A}[i] + 1$.[2] The sequence Ψ is differentially encoded, $\Psi(i + 1) - \Psi(i)$. Note that the Ψ values are increasing in the areas of $\mathcal{A}$ where the suffixes start with the same character c, because $cX < cY$ if and only if $X < Y$ in lexicographic order. It is enough to store those increasing values differentially with a method like Elias coding to achieve $O(nH_0)$ overall space [23]. Some additional information is stored to permit constant time access to Ψ. This includes the same C array used by the FM-index.

We are now ready to introduce the problem studied in this paper.

Definition 4 *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for each $1 \leq k \leq r$ and $\sum_{k=1}^{r} |T^k| = N$, where $T^1, T^2, \ldots, T^r$ contain overall s mutations 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 C?*

[2] Since $\mathcal{A}[1] = n$ because $T[n, n] = \$$ is the smallest suffix, it should hold $\mathcal{A}[\Psi(1)] = n + 1$. For technical convenience we set $\Psi(1)$ so that $\mathcal{A}[\Psi(1)] = 1$, which makes Ψ a permutation of $[1, n]$.

`locate`(P): *List the occurrence positions of P in C.*

`display`(k, i, j): *Return $T^k_{i,j}$.*

The basic collection indexing problem (without the requirement of the items to be mutated from each others) can be solved easily using the normal self-indexes for the concatenation $T^1\#T^2\#\cdots T^r\$$, where $\#$ is a special symbol not appearing in Σ. However, the space requirements achieved even with the high-entropy compressed indexes are 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 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 the sequel, we derive solutions whose space requirement depends on nH_k (instead of NH_k) and on s (instead of $o(N \log \sigma)$). We concentrate only on the `count`-query for succinctness. For the same reason, we restrict the analysis to mutations, although other types of variations are handled with the solutions as well.

3 Using Runs as a Complexity Measure

Self-repetitions are the fundamental source of redundancy in suffix arrays, enabling their compression. A self-repetition is a maximal interval $\mathcal{A}[i, i + l]$ of suffix array $\mathcal{A}$ having a *target interval* $\mathcal{A}[j, j + l]$ such that $\mathcal{A}[j + r] = \mathcal{A}[i + r] + 1$ for all $0 \leq r \leq l$. The intervals of Ψ corresponding to a self-repetition in the suffix array are called *runs*. As $\Psi(i) = \mathcal{A}^{-1}[\mathcal{A}[i] + 1]$, 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}. If the text is evident from the context, we will usually drop T and write just R, R_Ψ and R_{bwt}. There is a strong connection between the quantities R_Ψ and R_{bwt}, namely $R_\Psi \leq R \leq R_\Psi + \sigma$ [12], allowing to use them interchangeably under most circumstances. In addition to the trivial upper bound $R \leq N$, the bound $R \leq NH_k + \sigma^k$ for all k by Mäkinen and Navarro [12] is relevant for low entropy texts.

We will now prove some further bounds for texts obtained by repeating and mutating a basic sequence.

Definition 5 *Let $\#$ be a new character such that $\$ < \# < c$ for all $c \in \Sigma$. The r times repeated text $T = T_{1,n}$ is $\mathcal{T}^r = T^1T^2\cdots T^r$, where $T^r = T$ and $T^i = T_{1,n-1}\#$ for all $i < r$.*

Definition 6 *The context $C_{T,i}$ of suffix $T_{i,n}$ relative to text T is its shortest distinguishing prefix. Note that $C_{T,i}$ defines the position of $T_{i,n}$ in the suffix array of T. The significant prefix SP_{kn+i} of suffix $\mathcal{T}^r_{kn+i,N}$ is the context of $T_{i,n}$ relative to T.*

Theorem 7 *For all texts T and all $r \geq 1$, $R_\Psi(T) = R_\Psi(\mathcal{T}^r)$.*

Proof. Let $\mathcal{A}$ be the suffix array of T and $\mathcal{A}^r$ the suffix array of $\mathcal{T}^r$. The suffixes of $\mathcal{T}^r$ are first sorted by their significant prefixes. As $\$ < \#$, suffixes sharing the same significant prefix are further sorted by their starting positions in descending order. Hence

$$\mathcal{A}^r[r(i-1)+j] = (r-j)n + \mathcal{A}[i] \text{ for all } 1 \leq j \leq r.$$

By the definition of self-repetitions, $\mathcal{A}[i]$ and $\mathcal{A}[i+1]$ are contained in the same self-repetition of $\mathcal{A}$ if and only if the sequence $\mathcal{A}^r[r(i-1)+1, r(i+1)]$ is contained entirely in a self-repetition of $\mathcal{A}^r$. Hence there is one-to-one correspondence between the self-repetitions of $\mathcal{A}$ and $\mathcal{A}^r$. □

Next we proceed to *simple mutations* (single nucleotide polymorphisms), where a single character in $\mathcal{T}^r$ is randomly transformed into another character. Our goal is to bound the number of new runs created by the mutation.

Theorem 8 *Let $\mathcal{T}$ be a repeated text and $\mathcal{T}'$ the text created by transforming t_j into another character. Then $R_\Psi(T') \leq R_\Psi(T) + 2c + 1$, where c is the number of significant prefixes covering t_j.*

Proof. Let $\mathcal{A}$ be the suffix array of $\mathcal{T}$ and $\mathcal{A}'$ the suffix array of $\mathcal{T}'$. We call a suffix $\mathcal{T}'_{i,N}$ *moved* if SP_i is not its prefix. Hence the relative position of a moved prefix in $\mathcal{A}'$ differs from the position of the original prefix in $\mathcal{A}$. A moved prefix appearing inside a self-repetition of $\mathcal{A}$ or its target interval can break the self-repetition in two pieces in $\mathcal{A}'$. Additionally, each moved suffix may end up creating a new self-repetition by itself. As there are c moved suffixes, up to $2c+1$ new self-repetitions may be created in $\mathcal{A}'$.

The remaining suffixes are sorted by their significant prefixes. The mutation may affect the order of suffixes sharing the same significant prefix, yet does so in a consistent way creating no new self-repetitions. Hence a simple mutation can create no more than $2c+1$ new runs. □

3.1 Expected case bound

The above proof does not immediately generalize to multiple mutations. Note that we used two properties of significant prefixes: (i) SP_i is a prefix of $T_{i,N}$, and (ii) the suffixes sharing a significant prefix form a self-repetition in $\mathcal{A}$. To restore those properties after mutations, we need to update our set of prefixes. Unfortunately we have no easy way of bounding the length of such prefixes in the terms of original significant prefixes in the worst case. However, we can get meaningful expected case bounds.

Lemma 9 *Let $T = T_{1,n}$ be a random text. The expected length of the longest context in T is $O(\log_\sigma n)$.*

Proof. Let S^i be a prefix of $T_{i,n}$ of length l. For all $1 \leq i < j \leq n$ let X^i_j be the indicator variable for having S^i as a prefix of suffix $T_{j,n}$. If $i + l \leq j$, we have $\mathbf{E}[X^i_j] \leq \sigma^{-l}$. By the linearity of expectation, we have an upper bound of $\frac{1}{2}n^2\sigma^{-l}$ for the expected number of non-overlapping repeats of length l.

Consider now the case of where S^i appears as a prefix of $T_{j,n}$ for some $i < j < i + \frac{l}{2}$. Now S^i must be a repeating sequence with a period less than $\frac{l}{2}$. As there are at most $\sigma^{l/2}$ such sequences, we have an upper bound of $n\sigma^{-l/2}$ for the expected number of such sequences occurring as a substring of T.

Finally consider the case where the two occurrences overlap in at most $\frac{l}{2}$ positions. For such i and j, we have $\mathbf{E}[\sum_j X^i_j] \leq 2\sigma^{-l/2}$. Hence we have an upper bound of $2n\sigma^{-l/2}$ for the expected number of such repeats. By summing the bounds for $l = (2+x)\log_\sigma n$, we have

$$\mathbf{E}\left[\sum_{1 \leq i < j \leq n} X^i_j\right] \leq \frac{n^2}{2\sigma^l} + \frac{3n}{\sigma^{l/2}} \approx \frac{3}{n^{x/2}} = f(x).$$

By Markov's inequality, $f(x)$ is also a bound for the probability of having at least one repetition of length l. Hence the length of the longest repeat is at most $3\log_\sigma n$ for large enough n. □

Note that if we take a random text, repeat it r times and apply random mutations, the probability of getting a repeat of length l at different positions of the base sequence is at most r times the above bound. Hence the expected length of the longest significant prefix remains logarithmic in n and r.

Theorem 10 *Let $\mathcal{T}^r$ be the random text $T = T_{1,n}$ repeated r times. Let $\mathcal{S}^r$ be $\mathcal{T}^r$ after s simple mutations at random positions. The expected value of $R(\mathcal{S}^r)$ is at most $R(\mathcal{T}^r) + O(s\log_\sigma(rn))$.* □

3.2 Worst case bound

Although the number of runs can increase rapidly by mutations due to long distinguishing prefixes, we can obtain worst case bounds considering the k-th order entropy. Recall that

$$R(\mathcal{T}^r) = R(T) \leq nH_k(T) + \sigma^k = nH_k(\mathcal{T}^r) + \sigma^k.$$

Consider one mutation in $\mathcal{T}^r$ at position $pn + i$ producing mutated sequence $\mathcal{S}^r$. In the worst case, all the k-contexts $\mathcal{T}^r_{pn+i-j+1,pn+i-j+k}$ for $j = 0, 1, 2, \ldots, k$ change to $C_{pn+i-j} = \mathcal{S}^r_{pn+i-j+1,pn+i-j+k}$ such that s^r_{pn+i-j} does not yet appear in the context C_{pn+i-j} in T^r. It follows that $nH_k(\mathcal{S}^r) \leq nH_k(\mathcal{T}^r) + (k+1)\log\frac{N}{k+1}$ by elementary analysis of the differences in entropy formulas for $\mathcal{T}^r$ and $\mathcal{S}^r$. The case of s mutations follows in the obvious way.

Theorem 11 *Let $\mathcal{T}^r$ be the text T repeated r times. Let $\mathcal{S}^r$ be T^r after s simple mutations at random positions. The value of $R(\mathcal{S}^r)$ is at most $nH_k(T) + s(k+1)\log\frac{N}{k+1} + \sigma^k$.* □

3.3 Other types of mutations

The proof of Theorem 8 generalizes to different types of mutations. As long as no new material is inserted into the text, only those significant prefixes covering the beginning or the end of the mutation can create new runs. Hence the number of runs is quite robust with respect to the kinds mutations where the original sequence is cut in pieces and reassembled in an arbitrary order.

Theorem 12 *Let $\mathcal{T}$ be a repeated text and $\mathcal{T}'$ the text created by applying one complex mutation to T. Let m be the length of the mutation, c the number of significant prefixes of T covering the beginning of the mutation and d the number of such prefixes covering the end of the mutation.*

1. *For insertions, $R_\Psi(\mathcal{T}') \leq R_\Psi(\mathcal{T}) + 2(c+m) - 1$.*
2. *For deletions, $R_\Psi(\mathcal{T}') \leq R_\Psi(\mathcal{T}) + 2c + 1$.*
3. *For copies of existing substrings, $R_\Psi(\mathcal{T}') \leq R_\Psi(\mathcal{T}) + 2(c+d) + 2$.* □

4 Compressed Disk Suffix Array

Let us now describe the *Compressed Disk Suffix Array (CDSA)*, based on the CSA by Mäkinen, Navarro and Sadakane [13]. We use run-length encoding of the differences $\Psi(i)-\Psi(i-1)$ to store the array. To facilitate fast access to the array, we sample absolute $\Psi(i)$ values at a rate depending on the desired time-space trade-off. The resulting structure supports counting queries and allows an efficient secondary memory implementation.

To encode the run $\Psi(i)\Psi(i+1)\cdots\Psi(i+l)$, we write two integers: the gap after the previous run (or a sampled value) $\Psi(i)-\Psi(i-1)$ and the length of the run $l+1$. We use Elias delta coding to encode the integers. Let $b(p)$ be the binary representation of p. The encoding of the positive integer p is the binary string

$$\delta(p) = 0^{|b(t)|-1}b(t)b(p-2^{t-1}),$$

where $t = |b(p)|$. The length of the code is

$$|\delta(p)| = \log p + 2\log\log p - 2 = (1+o(1))\log p.$$

Let Ψ_c be the strictly increasing sequence of Ψ values corresponding to the suffixes of $T = T_{1,n}$ starting with character c. To bound the total length of codes, we note that the sum of differences $\Psi(i)-\Psi(i-1)$ inside each sequence Ψ_c is at most n. Hence the sum of all gaps between the runs of Ψ is at most σn. As the total length of all runs is n, the array takes

$$R\left(\log\frac{\sigma n}{R} + \log\frac{n}{R}\right)(1+o(1))$$

bits of space in the worst case. The bound is justified by the concavity of logarithm, making the worst case to be the one where the gaps are approximately $\frac{\sigma n}{R}$ and the lengths of runs are approximately $\frac{n}{R}$.

Let $|A|$ be the size of the array in bits. We build a higher level index by sampling the first $\Psi(i)$ value of each B-bit block of the array. As we start a new block whenever the first character of the suffix changes, we have $n_B \leq \frac{|A|}{B} + \sigma$ blocks in the array. As we need to write a few integers of size $\log n$ for each sample, the index takes $O(n_B \log n)$ bits of space.

We implement the counting queries using backward searching on Ψ. To find the first i with $\Psi(i)$ value at least sp or the last one with value at most ep corresponding to the endpoints of current interval $\mathcal{A}[sp, ep]$, we first use binary search on the index to find the correct block. This takes $O(\log n_B)$ time. When the correct block is found, we start to decode it from the beginning. Since decoding time is linear in B, the query takes $O(m(\log n_B + B))$ time. In a secondary memory implementation, we store the index in main memory and the array on disk. For each character of P except for the last one, we may have to retrieve up to two array blocks during the query. This gives us an upper bound of $2(m-1)$ disk accesses per query.

Theorem 13 *The CDSA for sequence $T = T_{1,n}$ requires*

$$|A| = R\left(\log\frac{\sigma n}{R} + \log\frac{n}{R}\right)(1+o(1))$$

bits of space for the array and $O(n_B \log n)$ bits of space for the index, where $n_B \leq \frac{|A|}{B} + \sigma$ is the number of blocks in the array. Counting queries take $O(m(\log n_B + B))$ time and require $2(m-1)$ disk accesses. □

5 Run-Length Encoded Wavelet Tree

Next we will describe how to construct space-efficient wavelet trees when considering a collection of multiple genomes. Results are achieved by a novel data structure that we call *Run-Length encoded Wavelet Tree*. We start by defining *rank* and *select* dictionaries for bit vectors, and the wavelet tree data structure for sequences.

Entropy-bound structures for bit vectors. For a bit vector B of length u, $rank_j(B, i)$ gives the number of j-bits in $B[1, i]$ for all $1 \leq i \leq u$ and $j \in \{0, 1\}$. The inverse function $select_j(B, x)$ gives the position of the x'th j-bit in the bit vector B. The *rank* and *select* queries can be solved in constant time using a succinct dictionary of size $uH_0(B) + o(u)$ [19, 21].

Entropy-bound structures for sequences. *Wavelet tree* [7] is a binary tree structure whose leaves represent the symbols in the alphabet. The root is associated with the whole sequence $T = T_{1,n}$. In a *balanced* wavelet tree, the left child (resp. right child) of the root is a wavelet tree of the sequence $T_<$ (resp. $T_\geq$) obtained by concatenating all positions i having $t_i < \sigma/2$ (resp. $t_i \geq \sigma/2$). 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. To recover a symbol t_i from the original sequence, we can traverse bit vectors of the wavelet tree starting from the root: In each internal node we choose either the left or the right subtree depending on the bit vector's i'th value. We set $i \leftarrow rank_0(B, i)$ when we choose the right subtree, and $i \leftarrow rank_1(B, i)$ otherwise. After $O(\log \sigma)$ recursive steps we arrive at the leaf node of the symbol c, and we know that the original $t_i = c$.

Let $T_{1,n}$ be an arbitrary sequence from the the alphabet Σ. Function $rank_c(T, i)$ gives the number of times the symbol c appears in the subsequence $T_{1,i}$. Function $select_c(T, x)$ is the inverse function of *rank*. The functions $rank_c(T, i)$ and $select_c(T, x)$ can be calculated from the balanced wavelet tree in $O(\log \sigma)$ recursive steps for any $c \in \Sigma$. For example $rank_c(T, i)$ can be solved by traversing the wavelet tree according to the symbol c: In each internal node, we go to the left subtree if $c < \sigma/2$ and right otherwise, and update i as in the previous paragraph. When we reach the leaf node of symbol c, the answer of the *rank* query is the value of i.

The space required by a balanced wavelet tree depends on how we encode the *rank* structures for the bit vectors. Entropy-bound dictionary structures for bit vectors [19, 21] can be used to represent the wavelet tree for an arbitrary sequence T in $nH_0(T)+o(n \log \sigma)$ bits [6]. However, when we are constructing the wavelet tree for a *BW-transformed* sequence T^{bwt}, a much better result can be achieved due to *implicit compression boosting* [14]: a wavelet tree for the sequence T^{bwt} requires only $nH_k(T)+o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$. This is a good result when considering only one sequence but for a collection of multiple sequences, that are almost identical, we notice a dependency on the overall length of the sequences. To make wavelet trees more suitable for these kind of collections, we describe a *Run-Length encoded Wavelet Tree* (RLWT) data structure, whose space requirement depends on the number of runs R instead of the overall length of the collection.

Run-Length encoded Wavelet Tree. Given a collection $\mathcal{C}$ and a concatenated sequence $T_{1,N}$ of all the sequences $T^i \in \mathcal{C}$, let R be the number of runs in the BW-transformed

sequence T^{bwt} of T. Let B^{all} be a level-wise concatenated bit vector of all the bit vectors in the balanced wavelet tree for the sequence T^{bwt}. In the worst case, each run in T^{bwt} equals one 0/1-bit run on every $\log\sigma$ levels of the wavelet tree, so that the upper-bound for the number of 0/1-bit runs in B^{all} is $R\log\sigma$. 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.

Theorem 14 ([9]) *Given a bit vector B of u bits containing b 1-bits, a* binary searchable dictionary representation *requires $gap(B) + O(b\log(u/b)/\log b) + O(b\log\log(u/b))$ bits of space and supports* rank *queries in $AT(u,b)$ time, where $AT(u,b)$ equals*

$$O\left(\min\left\{\sqrt{\frac{\log b}{\log\log b}}, \frac{\log\log u}{\log\log\log u}\cdot\log\log b, \log\log b + \frac{\log b}{\log\log u}\right\}\right),$$

and select *in $O(\log\log b)$ time. In the worst case, the* gap encoding measure *$gap(B)$ is roughly $b\log(u/b)$ 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 above theorem, the bit vectors can be represented in at most

$$R\log\sigma\log\frac{2N}{R} + O\left(\frac{R\log\sigma\log\frac{2N}{R}}{\log R + \log\log\sigma}\right) + O\left(R\log\sigma\log\log\frac{2N}{R}\right)$$

bits of space, where $\frac{N}{R} \leq \frac{r}{H_k}$ and $r = |\mathcal{C}|$. All the wavelet tree queries can be supported without storing the bit vector B^{all} itself. Next we will show how to calculate the *rank* queries on the bit vector B^{all} using only the bit vectors B^1 and B^{rl}.

Solving *rank* in RLWT. To calculate $rank_1(B^{all}, i)$ we first set $r \leftarrow rank_1(B^1, i)$, which is the number of 1-bit runs that start before or at the position i. If $r = 0$ then trivially $rank_1(B^{all}, i) = 0$. Let j be the starting position of the 1-bit run that precedes position i, to be exact $j \leftarrow select_1(B^1, r)$. From the definition of B^{rl} follows that the number of 1-bits before position j equals

$$rank_1(B^{all}, j-1) = \begin{cases} 0 & \text{if } r = 1, \\ select_1(B^{rl}, r-1) & \text{otherwise.} \end{cases}$$

The remaining part is 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 r'th 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 values $rank_1(B^{all}, j-1)$ and $rank_1(Ball, i) - rank_1(B^{all}, j-1)$ that we already calculated above. Solving *rank* for the bit vector B^{all} takes $AT(u,b) + O(\log\log b)$ time overall, and we get the following theorem for the RLWT structure.

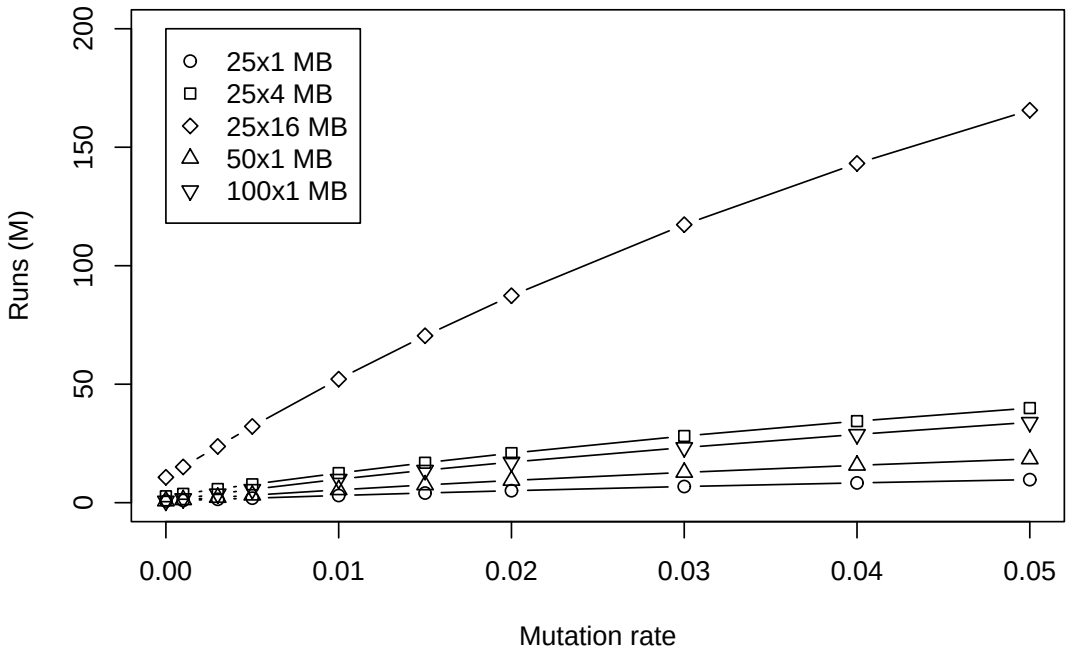


Figure 1: The number of runs in Ψ of repeated DNA sequences.

Theorem 15 *Given a collection $\mathcal{C}$ and a concatenated sequence T of all the sequences $T^i \in \mathcal{C}$, let R be the number of runs in the BW-transformed sequence T^{bwt} of T. The RLWT data structure for the sequence T^{bwt} takes*

$$R \log \sigma \log \frac{2N}{R}(1 + o(1)) + O\left(R \log \sigma \log \log \frac{2N}{R}\right)$$

bits of space, where $N = \|\mathcal{C}\|$. The queries $rank_c(T, i)$, $select_c(T, x)$ and the symbol T_i^{bwt}, for all $1 \leq i \leq N$, are solved in $O(\log \sigma \cdot (AT(u, b) + \log \log b))$ time, where $u \leq N \log \sigma$ and $b \leq \lceil \frac{1}{2} R \log \sigma \rceil$. $\square$

Backward search. Using the result from the above theorem with *backward searching* [4], we can count the number of occurrences of a pattern of length m in $O(m \log \sigma \cdot (AT(u, b) + \log \log b))$ time. The space required for counting queries is the same as in Theorem 15.

6 Experimental Results

We set our new self-indexes against the existing self-indexes Succinct Suffix Array (SSA) and Run-Length FM-index (RLFM) [12], where the former uses Huffman-shaped Wavelet tree directly on the BWT and the latter uses it on the run-length encoded BWT. We also compare to a self-index based on Lempel-Ziv parsing (LZ-index) [1] and to the alphabet-friendly FM-index (AFFM) [6] that requires $NH_k(\mathcal{C}) + o(N \log \sigma)$ bits. Furthermore, we

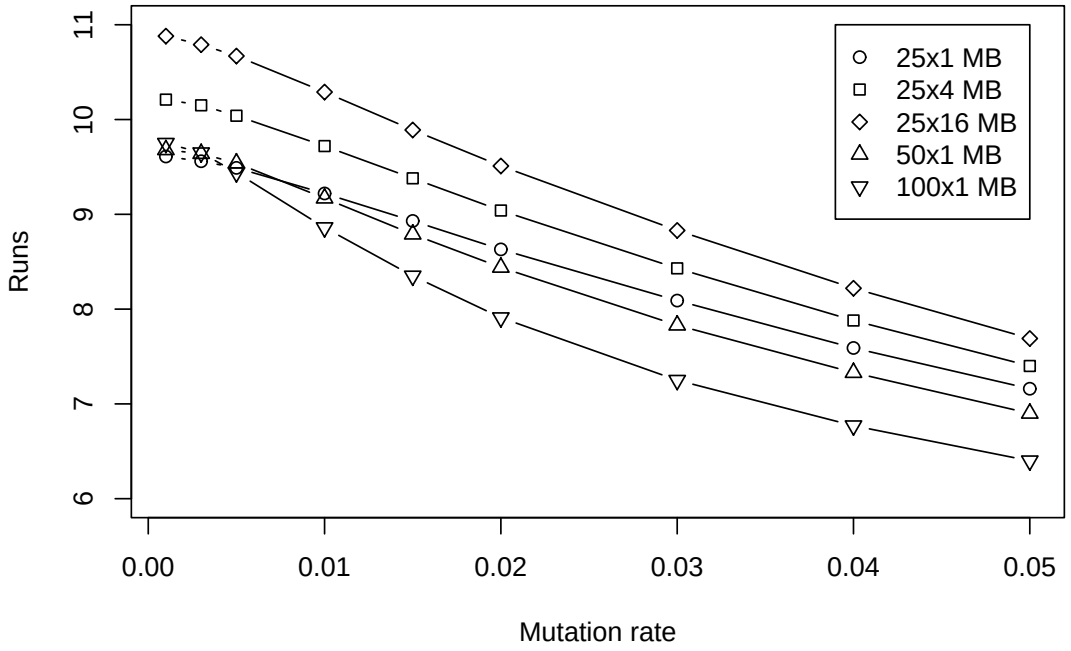


Figure 2: The number of new runs per mutation.

compare against a plain compressor to see how much we pay for the retrieval functionality. We use a highly efficient LZ77-based compressor `p7zip`[3] with options `-mx=9 -md=30`. The compressor uses a window of length up to 1 GB, and can thus compress texts with long repeats much better than standard Lempel-Ziv based compressors.

In our implementation of CDSA trivial runs of length 1 are encoded only by their gap values, whereas nontrivial runs of length $l+1$ are encoded by triples $(gap, 1, l)$. This saves one bit in trivial runs and wastes one bit in nontrivial ones. The trade-off is usually more noticeable when there are many trivial runs, making the average number of bits required to encode a run small. We use 32 kB block size for the CDSA unless otherwise noted. With such block size, the index size is less than 0.001 times the array size, making it suitable for secondary memory implementation.

Our implementation of the RLWT data structure uses simpler encoding for the bit vectors than in Theorem 15. The implemented structure solves counting queries for a pattern of length m in $O(m \log \sigma \cdot (\log(b/\log^2 b) + \log\log b))$ time, where $b \leq \lceil \frac{1}{2} R \log \sigma \rceil$.

For the tests, we use the DNA sequences from Pizza & Chili Corpus.[4] We take a 1, 4 or 16 MB prefix and repeat it 25, 50 and 100 times. Note that this is not exactly a repeated text of Definition 5, as we use no special characters to separate the basic sequences. Each character is randomly transformed into another character in {A,C,G,T} with probability corresponding to the mutation rate p. Characters in the first basic sequence are not mutated. The setting simulates the case of one base sequence and $r-1$ mutated sequences.

[3] `http://p7zip.sourceforge.net/`

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

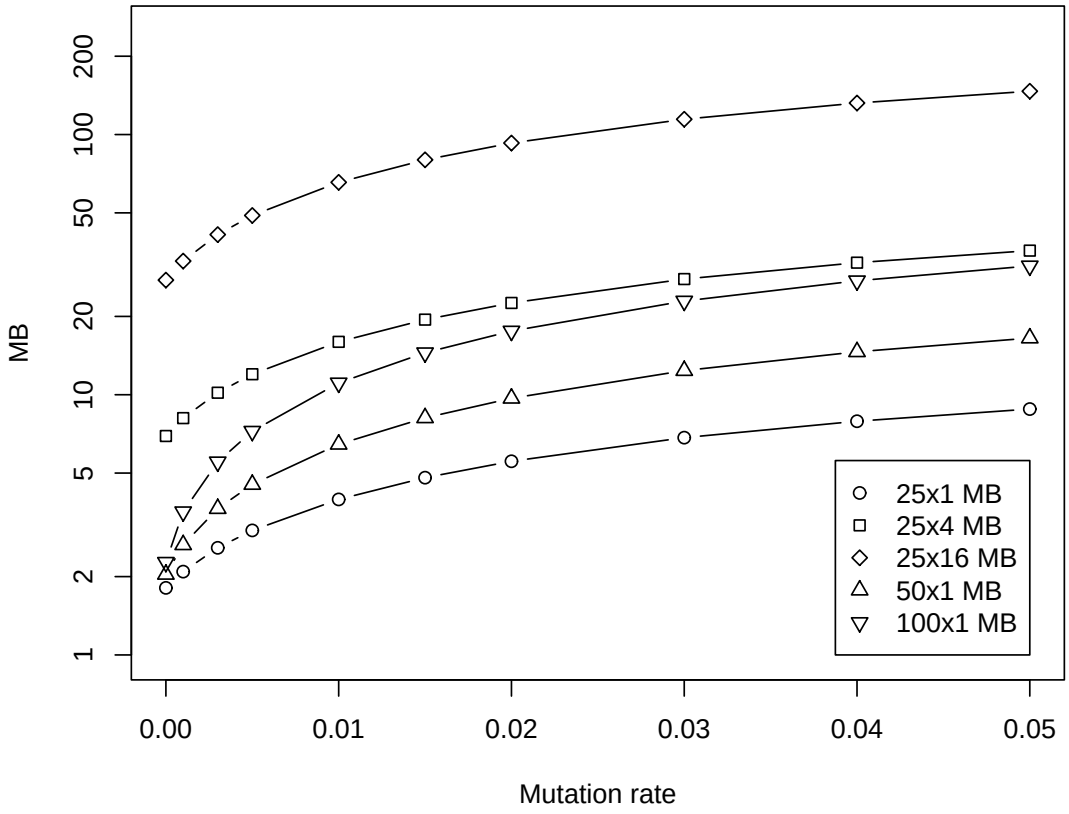


Figure 3: The size of CDSA on repeated DNA sequences.

Our other data set contains the source code for portable versions of OpenSSH[5]. We test our indexes on 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 contains multiple copies of same files as well as many highly similar files, making it highly compressible.

By Theorem 7, repetition of a base sequence does not increase the number of runs of Ψ. When the special characters separating the base sequences are removed, the results are slightly different. If there are R runs in Ψ of the base sequence, the number of runs in Ψ of a twice or more repeated sequence is $R + c$ for a (usually small) constant c. For example, there are $704,377$ runs in Ψ of the 1 MB DNA prefix, whereas the number of runs increases to $704,383$ when the sequence is repeated. The cause of this is that the last suffixes of other base sequences are often sorted differently than the corresponding last suffixes of the last base sequence.

Figure 1 shows the number of runs in Ψ of repeated DNA sequences. The number of runs grows somewhat sublinearly in the number of mutations. This is further elaborated in Figure 2. Note that the logarithmic dependence on the length of base sequence predicted

[5] `http://www.openssh.com/`

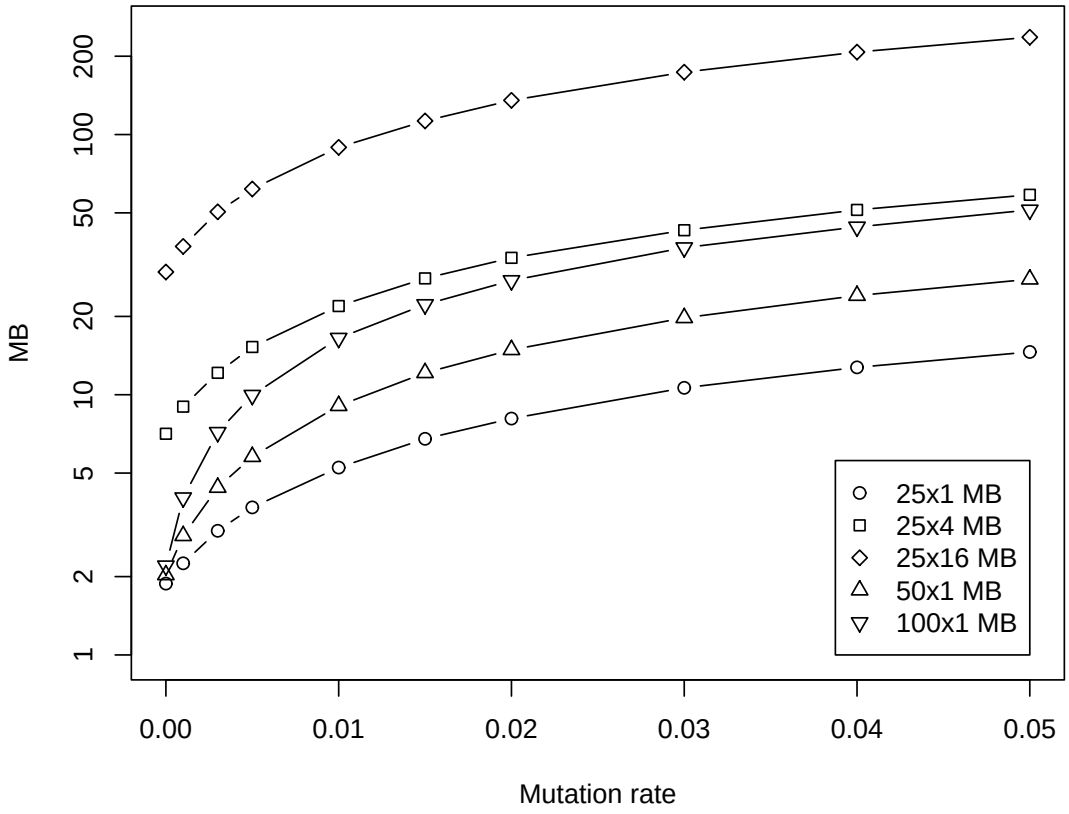


Figure 4: The size of RLWT on repeated DNA sequences.

for random texts in Theorem 10 also exhibits here. As the mutation rate or the number of repetitions increases, the probability of similar mutations increases, making it less likely for the moved suffixes to create new runs.

Figures 3 and 4 show the sizes of our structures on repeated DNA sequences. Both perform similarly, yet as the number of runs grows, CDSA outperforms RLWT, as predicted by the extra $\log \sigma$ factor in the space bounds.

We will now select the 25 times repeated 16 MB DNA prefix for further comparisons. As Figure 5 shows, our indexes clearly outperform the existing self-indexes when the number of mutations is small. Yet as Figure 6 shows, there might still be much room for improvement. To encode the sequence, `p7zip` requires approximately one tenth of the space. When the mutation rate is high, `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.

Next we compare our indexes with existing self-indexes as well as plain compressors on OpenSSH sources. In addition to `p7zip`, we use the well-known `gzip` and `bzip2` compressors with parameter `-9`. Due to their small block sizes, they give an idea of the traditional entropy-based compressibility of the collection. As seen in Figure 7, our indexes clearly outperform the existing self-indexes.

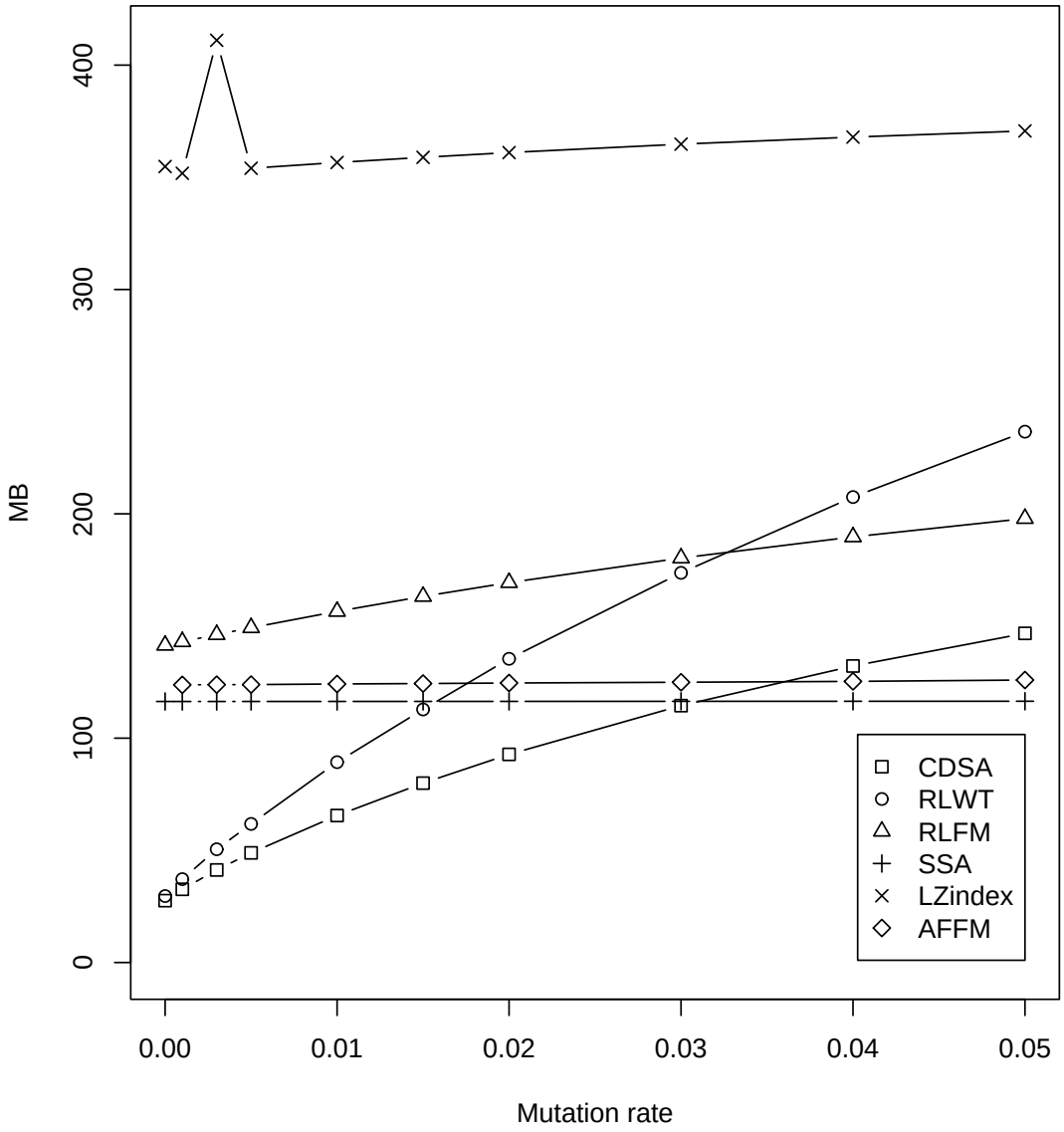


Figure 5: A comparison of our indexes with existing self-indexes.

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. The following table gives average query times and structure sizes on a 3 GHz Intel Pentium 4 Northwood machine with 3 GB RAM.

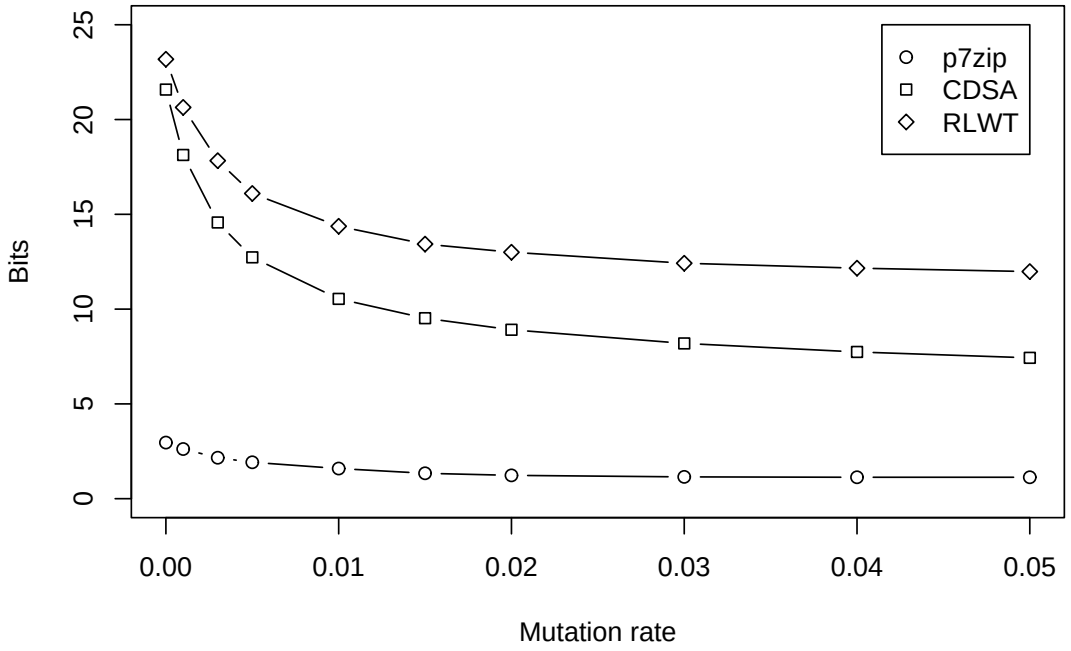


Figure 6: The average number of bits required to encode a run.

Structure	Time (μs)	Size (MB)
CDSA	188.0	71.51
RLWT	1050.0	89.30
RLFM	29.5	156.50
SSA	13.0	116.37
AFFM	19.4	124.15
LZ-index	79203.0	356.59

Note that we have used a main memory implementation of CDSA with 256 byte blocks here. Because of the smaller block size, the index is now almost 10 percents of the array size as opposed to less than 0.1 percent in the secondary memory implementation.

7 Discussion and Future Work

The new data structures proposed in this paper do not yet solve the full repetitive collection indexing problem, but rather give a first building block for such solution; we leave the `locate` and `display` queries for future research (a suitable sampling mechanism that takes advantage of the similarity of the texts to be indexed can be imagined to work). Also extending the solution to document retrieval queries [16, 25, 26], and building dynamic generalized suffix trees [24, 22] on top of the structure are imaginable directions for future studies. Maintaining a dynamic collection seems feasible by slight changes to the Run-Length encoded Wavelet Tree -structure.

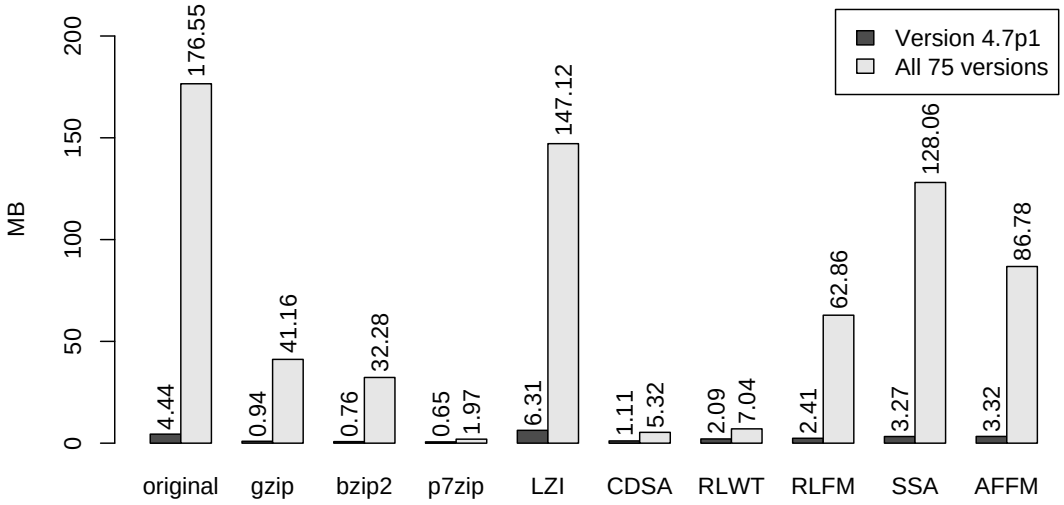


Figure 7: Compression results for OpenSSH sources.

In this study, we have only 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 [17, 18]. It is easy to see that the parsing of a repetitive text collection consist at most of $P(T^1) + s + 1$ phrases, where $P(T^1)$ gives the number of phrases in T^1. It follows that Lempel-Ziv based indexes like [17] require at most $O(n \log \sigma + s \log n)$ bits space. Hence, they are attractive for the application due to the good space bound; except that in practice our experiments indicate that the constant factors are quite large. Also, their functionality is limited to pattern searches and there does not seem to be a way to use them as building blocks of compressed suffix trees or alike structures.

Acknowledgments

Many of the ideas for the analysis part have had a great impact from the brainstorming with Gonzalo Navarro and Johannes Fischer when V. M. was visiting the University of Chile.

References

[1] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, LNCS 4009, pages 319–330. Springer-Verlag, 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 and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

[5] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal 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 Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[7] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[8] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[9] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. In *DCC '06: Proceedings of the Data Compression Conference (DCC'06)*, pages 213–222, 2006.

[10] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[11] N. Hall. Advanced sequencing technologies and their wider impact in microbiology. *The Journal of Experimental Biology*, 209:1518–1525, 2007.

[12] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[13] 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.

[14] Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE'07: Proceedings of the 14th Symposium on String Processing and Information Retrieval*, October 2007.

[15] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[16] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, pages 657–666, 2002.

[17] G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

[18] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[19] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[20] E. Pennisi. Breakthrough of the year: Human genetic variation. *Science*, 21:1842–1843, December 2007.

[21] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.

[22] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th LATIN*, LNCS, 2008. To appear.

[23] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[24] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2007. To appear. DOI 10.1007/s00224-006-1198-x.

[25] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.

[26] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, LNCS 4525, pages 217–228. Springer-Verlag, 2007.