# Storage and Retrieval of Individual Genomes (Extended Abstract)

Veli Mäkinen* Gonzalo Navarro† Jouni Sirén* Niko Välimäki*

**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. Flexible and efficient data analysis on a such typically huge collection is plausible using suffix trees. However, suffix tree occupies $O(N \log N)$ bits, which very soon inhibits in-memory analyses. Recent advances in full-text *self-indexing* reduce the space of suffix tree to $O(N \log \sigma)$ bits, where $\sigma$ is the alphabet size. In practice, the space reduction is more than 10-fold for example on suffix tree of Human Genome. However, this reduction remains a constant factor when more sequences are added to the collection

We develop a new self-index suited for the repetitive sequence collection setting. Its expected space requirement depends only on the length $n$ of the base sequence and the number $s$ of variations in its repeated copies. That is, the space reduction is no longer constant, but depends on $N/n$.

We believe the structure developed in this work will provide a fundamental basis for storage and retrieval of individual genomes as they become available due to rapid progress in the sequencing technologies.

## 1 Introduction

*Self-indexing* [9, 5, 24, 19] 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 approach becomes especially interesting when applied to dynamic collections of texts [3, 15].

A special case of a text collection is one which contains several *versions* of one or more *base sequences*. Such collections are not uncommon; consider for example the requirements for a standard *version control system*. An analogy to the storage and retrieval of version

*Department of Computer Science, University of Helsinki, Finland. {vmakinen,jltsiren,nvalimak}@cs.helsinki.fi. VM and NV funded by the Academy of Finland under grant 119815. JS funded by the Research Foundation of the University of Helsinki.

†Department of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl. Partially funded by Millennium Institute on Cell Dynamics and Biotechnology (ICDB), Grant P05-001-F, Mideplan, Chile.

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1674

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 [4, 12, 21]. With such data in hand, many fundamental issues become of top concern, like how to store, say, $10,000$ 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* [11] as that provides a variety of algorithmic tools to do analyzes 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 genome sequences as they 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 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 a new self-index, that is suitable for storing highly repetitive collections of texts, and a new compressed suffix tree based on it. Our scheme can also be thought of as a self-index for a given multiple alignment of a sequence collection, where one can retrieve any part of any sequence as well as make queries on the content of all sequences aligned. 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 new self-index presented in Sect. 4. Section 5 describes a new strategy to store *suffix array* samples, using and improving a classical solution for *persistent selection*. Section 6 shows how the newly derived structures can be applied to derive new compressed suffix trees. Section 7 discusses the extension of the abstract problem studied in this paper to the storage of *real* genomic sequence collections.

## 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 *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 use the standard notion of *empirical $k$-th order entropy* $H_k(T)$. For formal definition, see e.g. [18]. For our purposes, it is enough to know the basic property $H_k(T) \leq H_{k-1}(T) \leq \cdots \leq H_0(T) \leq \log \sigma$.

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 put the end marker $t_n \leftarrow \$$.

Let a *point mutation* (or just mutation for now) denote the event of a symbol changing into another symbol inside a string. We study the following problem.

**Definition 1** *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 $T^2, T^3, \ldots, T^r$ are mutated copies of the base sequence $T^1$ containing overall $s$ point mutations, 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 $C$?);* `locate`$(P)$ *(List the occurrence positions of $P$ in $C$); 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 only has one sequence $T$. 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 $\mathsf{SA}[1,n]$. 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 $\mathsf{SA}$ 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 Burrows-Wheeler transform. 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 $\mathsf{SA}$ and its inverse $\mathsf{SA}^{-1}$, 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 [19]. 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].

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 $\#$ is a special symbol not appearing in $\Sigma$. 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 with the collection of Def. 1 and 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 a solution whose space requirement depends on $nH_k$ (instead of $NH_k$) and on $s$ (instead of $o(N\log\sigma)$). Let us first consider a natural lower bound that takes into account these specific problem parameters. Consider a two-part compression scheme that first compresses $T^1$ with a high-order compressor and then the rest of the

sequences by encoding the mutations needed to convert each other sequence into $T^1$. The lower-bound for any such compressor is

$$nH_k(T^1) + \log \binom{N-n}{s} + s \log \sigma \approx nH_k(T^1) + s \log \frac{N}{s} + s \log \sigma \qquad (1)$$

where the first part is the lower bound of encoding $T^1$ with any high-order compressor, second part is the lower bound for telling the positions of the mutations among the $N-n$ possible, and third part is the lower bound for listing the $s$ mutations.

Finally, we show how to apply the collection indexes to turn the new *fully-compressed suffix trees* [7, 22] into a space expressed in the framework of the lower bound in Eq. 1.

We assume $\sigma = polylog(N)$ throughout the paper.

The abstract problem with point mutations studied here has nothing to do with the real variation occurring in genome sequences. However, all the techniques introduced can be extended to the full set of mutation events. This will be discussed in Sect. 7.

# 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 $\mathsf{SA}[i, i+l]$ of suffix array $\mathsf{SA}$ 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 $\Psi(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]+1]$. The intervals of $\Psi$ 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 $\Psi$ of text $T$ and $R(T) = R_{bwt}(T)$ the number of equal letter runs in $T_{bwt}$. There is a strong connection between the quantities $R_\Psi$ and $R_{bwt}$, namely $R_\Psi \leq R \leq R_\Psi + \sigma$ [14], 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 [14] is relevant for low entropy texts.

We will now prove some further bounds for texts obtained by repeating and mutating substrings of a base sequence. To simplify the analysis, we add a new character # such that $\# < \$ < c$ for all $c \in \Sigma$. Furthermore, we assume that the ordering between two occurrences of character # is decided by their positions in the sequence, making each occurrence of # a different character in practice.

**Definition 2** *The $r$ times* repeated collection *of base text $T = T_{1,n}$ is $\mathcal{T}^r = T^1 T^2 \cdots T^r$, where $T^r = T$ and $T^i = T_{1,n-1}\#$ for all $i < r$.*

**Definition 3** *The* context *$C_{T,i}$ of suffix $T_{i,n}$ relative to text $T$ is its* shortest distinguishing prefix*, i.e., the prefix that does not appear anywhere else in $T_{1,n}$ as a substring. Note that $C_{T,i}$ defines the position of $T_{i,n}$ in the suffix array of $T$.*

**Definition 4** *Let $\mathcal{T}^r$ be a collection of $r$ texts, each derived by mutations from a base sequence $T$. A* significant prefix *$SP_{i,j}$ of the suffix starting at position $j$ of sequence $T^i$ is the shortest prefix not occurring anywhere else in $\mathcal{T}^r$ as a substring* except *possibly as a prefix of some $T^k_{j,n}$, $k \neq j$.*

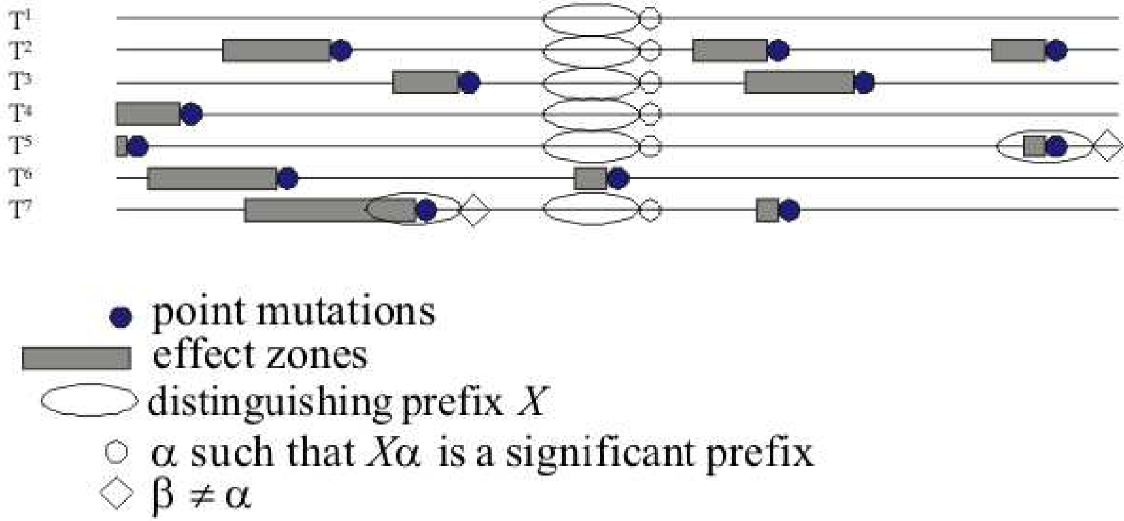


Figure 1: An example of distinguishing and significant prefix concepts. Text $T^1$ contains a distinguishing prefix $X$ which is repeated in its mutated copies $T^2, T^3, T^4, T^5$, and $T^7$. Text $T^6$ has a mutation inside $X$. Due to other mutations, texts $T^5$ and $T^7$ now contain $X$ in some other positions, and hence $X$ is not a significant prefix. However, extending $X$ with string $\alpha$ makes $X\alpha$ unique to the original position of $X$, while the other two occurrences of $X$ are succeeded by string $\beta \neq \alpha$. Hence, $X\alpha$ is a significant prefix, for $\alpha$ being the shortest extension having the required property. The effect zones illustrate the positions where the significant prefixes are affected by the mutations.

Figure 1 illustrates the definitions.

We show some basic results concerning the number of runs in repeated and mutated texts. Proofs appear in the full version of the paper. Expected case proofs extend those in [27, pp.263–265].

**Lemma 5** *For all texts $T$ and all $r \geq 1$, $R_\Psi(T) = R_\Psi(\mathcal{T}^r)$.*

*Proof.* (Sketch) All suffixes corresponding to the same suffix of $T$ are grouped together in the suffix array of $\mathcal{T}^r$. Each group is further ordered from the suffix of $T^1$ to the suffix of $T^r$. Hence there is one-to-one correspondence between the self-repetitions of suffix arrays of $T$ and $\mathcal{T}^r$. □

**Lemma 6** *Let $\mathcal{T}^r = T^1 T^2 \cdots T^r$ be a repeated collection and $\mathcal{T}'$ the collection created by transforming $t^i_j$, for some $1 < i \leq r$ and $1 \leq j < n$, into another character. Then $R_\Psi(\mathcal{T}') \leq R_\Psi(\mathcal{T}^r) + 2c + 1 = R_\Psi(T) + 2c + 1$, where $c$ is the number of significant prefixes covering $t^i_j$.*

*Proof.* (Sketch) The relative position of a suffix in the suffix array can change only if its significant prefix has mutated. Each such suffix can interfere with a constant number of runs. The ordering of suffixes sharing a significant prefix can change, but this does not create additional runs. □

**Lemma 7** *Let $T = T_{1,n}$ be a random text. The expected length of the longest context is $O(\log_\sigma n)$.*

*Proof.* (Sketch) The expected number of non-overlapping repeats of length $l$ is $O(n^2/\sigma^l)$. Markov's inequality bounds the probability of having such a repeat of length $c \cdot \log_\sigma n$ exponentially in $c^{-1}$. Overlapping repeats are handled in a similar manner. □

**Lemma 8** *Let $\mathcal{T}^r$ be the repeated collection of random text $T = T_{1,n}$ with total length $N = nr$. Let $\mathcal{S}^r$ be $\mathcal{T}^r$ after $s$ point mutations at random positions in $T^2T^3 \cdots T^r$. The expected value of $R_\Psi(\mathcal{S}^r)$ is at most $R_\Psi(T) + O(s \log_\sigma N)$.*

# 4 Run-Length FM-index for Repetitive Collections

Recall that FM-index requires table $C$ and function $rank_c(L,i)$ on the Burrows-Wheeler transform $L = T^{bwt}$ to support pattern search. The *Run-Length FM-Index (RLFM)* [14] uses a reduction such that $L[1,N]$ is replaced by $L'[1,R]$, where equal-letter runs are compressed into one symbol. Two bit-vectors $B$ and $B'$ of length $N$, each having $R$ bits set, and a table $C_B$ analogous to $C$, are stored as well. Using *rank* and *select* on these bit-vectors and on $L'$ is enough to simulate the corresponding operations on $L$ [14], where $select_c(X,j) = i$ is such that $rank_c(X,i) = j$ and $X[i] = c$

The original proposal [14] uses $2N + o(N)$ bits for $B$ and $B'$ to support constant time binary *rank* and *select*. Then (negligible) $2\sigma \log N$ bits are used for $C$ and $C_B$, and $R \log \sigma (1 + o(1))$ bits for the *wavelet tree* [8, 6] of $L'$ to support constant time $rank_c()$ under *polylog*$(N)$ size alphabets.

The space can be improved using the so-called BSD representation for $B$ and $B'$:

**Lemma 9 ([10])** *Given a bit vector $B$ of $u$ bits containing $b$ 1-bits, a* binary searchable dictionary representation *(BSD) 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 $AT(u,b)$ 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, length of the* gap encoded sequence *$|gap(B)|$ is $b\log(u/b) + O(b\log\log(u/b))$ bits.* □

We have immediately the following result, by noticing that for us $b = R$ and $u = N$.

**Lemma 10** *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 RLFM data structure for the sequence $T^{bwt}$ can be represented in $(R\log\sigma + 2R\log\frac{N}{R})(1+o(1)) + O\left(R\log\log\frac{N}{R}\right)$ bits of space. The queries $rank_c(T^{bwt},i)$, $select_c(T^{bwt},x)$ and retrieving the symbol $t_i^{bwt}$, for all $1 \le i \le N$, are solved in $O(t_{\mathsf{LF}})$ time. In particular, the structure supports $\boldsymbol{count}(P)$ in time $O(|P| t_{\mathsf{LF}})$, where $t_{\mathsf{LF}} = t_{\mathsf{AT}} = AT(N,R)$.* □

## 5 Suffix Array Samples

To support the other two functions of the repetitive collection indexing problem, namely, `display`() and `locate`(), we need to be able to map the suffixes of the text into suffix array indexes and vice versa. The standard solution [19] in self-indexes is to sample every $d$-th suffix of each text in the collection in an array $D[1, N/d+1]$, such that $D[i] = \mathsf{SA}^{-1}(id)$, mark the locations $D[i]$ into a bit-vector $B[1, N]$, such that $B[D[i]] = 1$ for all $1 \leq i \leq N/d+1$, and store the samples in the suffix array order in a table $S[1, rank_1(B, N)]$, such that $S[rank_1(B, D[i])] = id$.

Then $\texttt{display}(k, i, j)$ works as follows. Let $SP[k]$ be the starting position of $T^k$ in the concatenated sequence $T = T^1 T^2 \cdots T^r$. Value $D[(SP[k]+j)/d+2] = e$ tells us that the nearest sampled suffix after $T_{SP[k]+j,N}$ is stored at suffix array index $\mathsf{SA}[e]$. Following $LF$-mapping starting at position $e$ reveals us backwards a substring that covers $T^k_{i,j}$ in time $O(t_{\mathsf{LF}}(d+j-i+1))$.

Function `locate(P)` works in a similar fashion; first backward search is applied to find the range $\mathsf{SA}[sp, ep]$ containing the occurrences of the pattern $P$ and $\mathsf{SA}[i]$ is computed for each $sp \leq i \leq ep$ as follows. If suffix $\mathsf{SA}[i]$ is not sampled ($B[i] = 0$), then $LF$-mapping is applied until an index $j$ is found where $\mathsf{SA}[j]$ is sampled ($B[j] = 1$). Then $\mathsf{SA}[i] = S[rank_1(B, j)] + c$, where $c < d$ is the number of times $LF$-mapping was applied. This takes time $t_{\mathsf{SA}} = O(t_{\mathsf{LF}} d)$.

The space required by the standard solution is $O((N/d)\log N + N)$ bits, which can be reduced to $O((N/d)\log N)$ by using Theorem 9; this changes the time for `locate`() into $t_{\mathsf{SA}} = O((t_{\mathsf{LF}} + t_{\mathsf{AT}})d)$, where $t_{\mathsf{AT}} = AT(N, N/d)$.

Our objective is to have all time requirements in $O(polylog(N))$ which holds only with the above approaches if we assume $r = O(polylog(N))$; then $d$ can be chosen as $r \log N$ to make $O((N/d)\log N) = O(n)$, i.e., independent of $N$ as we wish.

### 5.1 Improving Space for `display`()

We will store samples only for $T^1$, that is, table $D[1, n/d+1]$ has the suffix array entry of every $d$-th suffix $T^1_{di,n}$ stored at $D[i] = \mathsf{SA}^{-1}(id)$.

To be able to use the same samples for other texts in the collection, we mark the locations of mutations into bit-vectors. Let $M^k[1, |T^k|]$ be a bit-vector where the locations of the mutations inside $T^k$ are marked. The mutated symbols are stored in another array $MS^k[1, rank_1(M^k, |T^k|)]$ in their order of occurrence in $T^k$.

Consider now a query $\texttt{display}(k, i, j)$. The substring $T^1_{i,j}$ is extracted using the samples just like in the standard approach. It is easy to see that while extracting $T^1_{i,j}$, the mutations stored for $T^k$ can also be extracted using $rank$-function on $M^k$. Table $MS^k$ occupies overall $s \log \sigma$ bits. Bit-vectors $M^k$ can be represented using Theorem 9 in overall $s \log \frac{N-n}{s}(1+o(1)) + O\left(s \log\log \frac{N-n}{s}\right)$ bits. What we gain is that $O((N/d)\log N)$ becomes $O((n/d)\log n)$.

### 5.2 Improving Space for `locate`()

We use the same strategy as for `display`(), sampling only $T^1$ regularly, but this time we need to sample also parts of the other texts as discussed next.

Let us first consider the case of $r$-identical texts. We know that the suffixes $T^1_{p,n}, T^2_{p,n}, \ldots, T^r_{p,n}$ will all be consecutive and in the same order in $\mathsf{SA}$. Hence, once every $d$-th suffix of $T^1$ is sampled, we can reveal any $\mathsf{SA}[i]$ by applying $LF$-mapping at most $d$ times until finding an entry $j$ such that $\mathsf{SA}[j']$ is sampled for some $j' < j$ and $j - j' \leq r$. Checking whether this is the case is identical to $j - select_1(B, rank_1(B, j)) \leq r$, where $B$ is the bit-vector marking the locations of the sampled suffixes of $T^1$ in $\mathsf{SA}$. Then $\mathsf{SA}[j]$ corresponds to suffix $T^k_{S[rank_1(B,j')]+c,n}$, where $S$ is the table storing the sampled suffixes in the order they appear in $\mathsf{SA}$, $c < d$ is the number of times $LF$-mapping was applied, and $k = j - select_1(B, rank_1(B, j)) + 1$.

Generalizing the scheme to work under mutations is non-trivial. We introduce a strategy that splits the suffixes into two classes A and B such that class A suffixes are computed via $T^1$ samples and for class B we add new samples from all the texts. Recall Lemma 6; Class B contains the $c$ suffixes whose significant prefixes overlap one or more mutations. Class A contains all other suffixes.

Let us first consider the case when $\mathsf{SA}[i]$ is a class B suffix. Class B suffixes form at most $s$ disjoint regions in texts $T^k$, $2 \leq k \leq r$. We sample every $d$-th suffix inside each of these regions. The suffix array indexes containing these sampled suffixes are marked in a bit-vector $E[1, N]$, and a table $S^B[1, rank_1(E, N)]$ stores these sampled suffixes in the order they appear in $\mathsf{SA}$. Retrieving $\mathsf{SA}[i]$ is completely analogous to the standard sampling scheme by using $S^B$ in place of $S$ and $E$ in place of $B$. The space is bounded by $O((c/d) \log N)$, which is $O(((s \log_\sigma N)/d) \log N)$ in the average case.

Computing $\mathsf{SA}[i]$ for class A suffixes is more challenging than in the case of $r$ identical texts when all suffixes were class A. The problem can be divided into the following subproblems: (i) Not all sampled suffixes of $T^1$ will have counterparts in all the other texts. Hence, we need to store explicitly a list $Q[rank_1(B, \mathsf{SA}^{-1}[id])] = k_1 k_2 \cdots k_p$ denoting texts $T^{k_1}, T^{k_2}, \ldots, T^{k_p}$, $p \leq r$, that correspond to a sampled suffix $T^1_{id,n}$. However, this takes too much space. (ii) Class B suffixes break the order of the suffixes aligned to the same sampled $T^1$ suffix, making it difficult to know, once at $\mathsf{SA}[j]$, whether there is a sampled suffix of $T^1$ at some position $\mathsf{SA}[j']$ close enough.

Let us consider subproblem (ii) first. The solution is to explicitly mark all class B suffixes in $\mathsf{SA}$ into a bit-vector $F[1, N]$, and to store for each sampled suffix $T^1_{id,n}$ its lexicographic *rank* $e$ among the suffixes in the list $Q[rank_1(B, \mathsf{SA}^{-1}[id])] = k_1 k_2 \cdots k_p$, that is, $e$ such that $k_e = 1$. Now, consider again the situation where $\mathsf{SA}[i]$ belongs to class A and $LF$ mapping has brought us to entry $\mathsf{SA}[j]$. Let us compute $prev = select_1(B, rank_1(B, j))$, $succ = select_1(B, rank_1(B, j) + 1)$, $dprev = (j - prev) - (rank_1(F, j) - rank_1(F, prev))$, and $dsucc = succ - j - (rank_1(F, succ) - rank_1(F, j))$. Let $Q[rank_1(B, prev)] = k_1 k_2 \cdots k_p$ and $e$ be such that $k_e = 1$. If $dprev \leq p - e$ then $k_{e+dprev}$ is the number of the text where suffix $\mathsf{SA}[j]$ belongs to. This follows from the fact that the effect of class B suffixes is eliminated using *rank*, so it remains to calculate how many class A suffixes there are between the sampled suffix and current position. If this number is smaller than (or equal to the) the number of suffixes with rank higher than that of $\mathsf{SA}[prev]$ in the list $Q[rank_1(B, prev)]$, then (and only then) $\mathsf{SA}[j]$ belongs to the same list. Analogously, one can check whether $\mathsf{SA}[j]$ belongs to the list $Q[rank_1(B, succ)] = k_1 k_2 \cdots k_{p'}$ of $\mathsf{SA}[succ]$. After at most $d$-steps of $LF$-mapping the correct $Q$-list is found. The additional space needed is $O(c \log \frac{N-n}{c})$ bits for the BSD of bit-vector $F$.

Finally, we are left with subproblem (i): the lists $Q[1], Q[2], \ldots, Q[n/d]$ occupy in total $O((n/d)r \log r)$ bits. We will next improve the space to $O(s \log s)$ bits modifying a classical solution by Overmars [20] to $k^{th}$ *element/rank searching in the past*. The proof of Theorem 12 reviews the original structure and the proof of Theorem 13 shows how to make it more space-efficient and *confluental persistent* (see [13] for background).

**Definition 11** *Let $E(t) = e_1^t e_2^t \cdots e_{p_t}^t \in \mathcal{R}^* = \{1, 2, \ldots, r\}^*$ be a sequence of* elements *at* time point *$t \in H$, where $H \subseteq \mathcal{H} = \{1, 2, \ldots, h\}$, such that $E(t)$ can be constructed from $E(tprev)$, $tprev = max\{t' \in H \mid t' < t\}$, by* deleting *some $e_k^{tprev}$ or* inserting *a new element $e \in \mathcal{R}$ between some $e_{k-1}^{tprev}$ and $e_k^{tprev}$ The* persistent selection *problem is to construct a static data structure $\mathcal{D}$ on $\{E(t) | t \in H\}$ that supports operation $select(t, k) = e_k^t$. The* online persistent selection *problem is to maintain $\mathcal{D}$ such that it supports $insert(t, e, k)$ and $delete(t, k)$, where value $t$ must be at least* $\max(H)$*; The* confluental persistent selection *problem allows value $t$ to be any $t \in \mathcal{H}$ also for insertions and deletions;*

**Theorem 12 ([20])** *There is a data structure $\mathcal{D}$ for the online persistent selection problem occupying $O(x(\log x \log h + \log r))$ bits of space and supporting $select(t, k)$ in $O(\log x)$ time, and $insert(t, e, k)$ and $delete(t, k)$ in amortized $O(\log x)$ time, where $x$ is the number of insertion and deletion operations executed during the lifetime of $\mathcal{D}$.*

*Proof.* (Sketch) The structure $\mathcal{D}$ is a variant of balanced binary tree that stores subtree sizes in its internal nodes, enhanced with *path copying* and *fractional cascading* to support persistance: Consider a tree $\mathcal{T}(t)$ for storing elements of $E(t)$ in its leaves and having subtree sizes stored in its internal nodes. Selecting the $k$-th leaf equals accessing $e_k^t$. It is easy to find that leaf by following the path from the root and comparing $k$ with the sum of subtree sizes of nodes that remain hanging left side of the path; if at node $v$ the current sum plus subtree size of the left child of $v$ is smaller than $k$, go right, otherwise go left. Now, consider an insertion to produce $E(t)$ from $E(tprev)$. To produce $\mathcal{T}(t)$ one can add a new leaf to $\mathcal{T}(tprev)$ and increment the subtree sizes by one on the path to the new leaf. To make this change persistent, the idea in [20] is to copy the old subtree size information into a new field on each node on the path and increment that. The field is labeled with the time $t$ and also pointers are associated to the corresponding fields on the left and right child of the node, respectively. Here corresponding means a field whose time-stamp is largest $t'$ such that $t' \leq t$. Analogous procedure is executed for deletions, except that the corresponding leaf is not deleted, but only the subtree sizes are updated accordingly. This procedure is repeated over all time points and the tree is rebalanced when necessary. The rotations to rebalance the tree require merging the lists of fields storing the time-stamped information. The cost of rebalancing can be amortized over insertions and deletions [20]. The root of the tree stores the time-stamped list as a binary search tree to provide $O(\log x)$ time access to the entries. The required space for the tree itself is $O(x \log x \log h)$ bits as each of the $x$ updates creates a new field occupying $O(\log h)$ bits for each of the $O(\log x)$ nodes on the path from root to the leaf. In addition, each leaf contains a value of size $\log r$ bits. □

**Theorem 13** *There is a data structure $\mathcal{D}$ for the persistent selection problem occupying $O(x(\log x + \log h + \log r))$ bits of space and supporting $select(t, k)$ in $O(\log x)$ time. There*

*is also an online/confluental version of $\mathcal{D}$ that occupies the same space, but select$(t,k)$ takes $O(\log^2 x)$ time, and insert$(t,e,k)$ and delete$(t,k)$ take amortized $O(\log^2 x)$ time.*

*Proof.* We modify the structure of Theorem 12 by replacing the time-stamped lists of fields in each node of the tree with two partial sums that can be represented succinctly. Let $S^v = s^v_0 s^v_1 s^v_2, \cdots s^v_{k^v}$ be the list of subtree sizes stored in some node $v$, where $s^v_0 = 0$. Let $\hat{S}^v = (s^v_1 - s^v_0)(s^v_2 - s^v_1)\cdots(s^v_{k^v} - s^v_{k^v-1})$. We represent $\hat{S}^v$ via succinct data structure for (dynamic) partial sums to support operations $select(\hat{S}^v, i) = \sum_{j=1}^{i} \hat{s}^v_j = s^v_i$. In addition, we construct a bit-vector $B^v[1,k^v]$ where $B^v[i] = 1$ if and only if the change $s^v_i$ came from the right child of $v$. Notice that we do not need the explicit fractional cascading links anymore, as we have the connection $select(\hat{S}^v, i) = select(\hat{S}^l, i-i') + select(\hat{S}^r, i')$, where l-¿r, right? $i' = rank_1(B^v, i)$, and $l$ and $r$ are the left and right children of $v$. That is, $select(\hat{S}^l, i-i')$ and $select(\hat{S}^l, i')$ are the subtree sizes of nodes $l$ and $r$, respectively, at the same time point as $s^v_i$. In the root of the tree we keep the original binary search tree to map the parameter $t$ to its rank $i$ and after that the formulas above can be used to compare subtree sizes to value of parameter $k$. Notice also that confluental *insert* and *delete* are immediately provided if we can support dynamic *select* on $\hat{S}^v$ and dynamic *rank* on $B^v$.

Let us consider how to provide $select(\hat{S}^v, i) = s^v_i$. First notice that $\sum_{v \in \mathcal{T}} \sum_{j=1}^{k^v} \hat{s}^v_j = O(x \log x)$ because each insertion or deletion changes the subtree size by one on $O(\log x)$ nodes. Hence, we can afford to use unary coding for these values. We represent each $\hat{S}^v$ by a bit-vector $F^v = f(\hat{s}^v_1) f(\hat{s}^v_2) \cdots f(\hat{s}^v_{k^v})$, where $f(x) = 1^x$ if $x > 0$ otherwise $f(x) = 0^{-x}$, and by a bit-vector $G = 10^{|f(\hat{s}^v_1)|-1} 10^{|f(\hat{s}^v_2)|-1} \cdots 10^{|f(\hat{s}^v_{k^v})|-1}$. Then $select(\hat{S}^v, i)$ equals $2 \cdot rank_1(F^v, j-1) - (j-1)$, where $j = select_1(G^v, i+1)$. That is, $\sum_{v \in \mathcal{T}} (|F^v| + |G|)(1 + o(1)) = O(x \log x)$ bits is enough to support constant time *select* on all subtree sizes, when the tree is static. In the dynamic case, *select* takes $O(\log x)$ time [1]. Same analysis holds for bit-vectors $B^v$.

In summary, the tree in the root takes $O(x \log h)$ bits, and support *rank* for $t$ in $O(\log x)$ time. The bit-vectors in the main tree occupy $O(x \log x)$ bits and make a slowdown of $O(1)$ or $O(\log x)$ per node depending on the case. The associated values in the leaves occupy $O(x \log r)$ bits. □

Combining Lemmas 8 and 10 with Theorem 13 applied to sampling gives us the main result of the paper:

**Theorem 14** *Given a collection $\mathcal{C}$ and a concatenated sequence $T$ of all the $r$ sequences $T^i \in \mathcal{C}$, there is a data structure for the* repetitive collection problem *taking*

$$(R \log \sigma + 2R \log \frac{N}{R})(1 + o(1)) + O\left(R \log\log \frac{N}{R}\right)$$
$$+O(s \log_\sigma N \log \frac{N}{s \log_\sigma N}) + O(s \log s) + O(r \log N)$$
$$+O(((s \log_\sigma N)/d) \log N) + O((n/d) \log n)$$

*bits of space in the average case. The structure supports* $\mathtt{count}(P)$ *in time* $O(|P| t_{\mathsf{LF}})$, $\mathtt{locate}(P)$ *in time of* $\mathtt{count}(P)$ *plus* $O(d(t_{\mathsf{LF}} + t_{\mathsf{AT}}) + \log s)$ *per occurrence,* $\mathtt{display}(k,i,j)$ *in time* $O((d + j - i + 1)(t_{\mathsf{LF}} + t_{\mathsf{AT}})$*, computing* $\mathsf{SA}[i]$ *and* $\mathsf{SA}^{-1}[(k,j)]$ *in time* $t_{\mathsf{SA}} = O(d(t_{\mathsf{LF}} + t_{\mathsf{AT}}) + \log s)$*, and* $T(SA[i])$ *in time* $O(AT(N,\sigma))$*, where* $t_{\mathsf{LF}} = AT(N,R)$*.*

*Proof.* (Sketch) The discussion preceding persistent selection developed data structures occupying $O(s \log_\sigma N \log \frac{N}{s \log_\sigma N}) + O(((s \log_\sigma N)/d) \log N)$ bits to support parts of the remaining *locate*() operation. These are larger than the ones for *display*(). Theorem 13 provides a solution to the subproblem (i): we can replace the lists $Q[1], Q[2], \ldots, Q[n/d]$ by persistent select, where the $s$ mutations cause insertions and deletions to the structure (as they change the rank of a text between two samples). There will be $s$ such updates, and on any given position $i$ of the text $T^1$ (including those that are sampled) one can select the $k$-th text aligned to that suffix in $O(\log s)$ time. The space usage of this persistent structure is $O(s \log s)$. Finally, $\mathsf{SA}[i]$ computation is identical to *locate*(), but computation of $\mathsf{SA}^{-1}[(k,j)]$ is not yet supported. Computation of $\mathsf{SA}^{-1}[(k,j)]$ resembles the display operation in the case $t_j^k$ belongs to an area where a sampled position is at distance $d$. Otherwise one must follow closest sampled position after $t_j^1$ to suffix array, and use at most $d$ times the $LF$-mapping to find out $\mathsf{SA}^{-1}[(1,j)]$. Now, to find the rank of text $T^k$ with respect to that of text $T^1$ in the persistent tree of Theorem 13 storing the lexicographic order of suffixes aligned to position $j$, one can do the following. Whenever a new leaf is added to the persistent tree, associate to that text position a pointer to this leaf. These pointers can be stored in $O(s \log s)$ bits and their locations can be marked using $s \log \frac{N}{s}(1 + o(1))$ space, so that one can find the closest location to $(k, j)$ having a pointer, using $rank$ in $t_{\mathsf{AT}}$ time. Following this pointer to the persistent tree leaf, and continuing to the root of the tree (and back), one can compute the rank of the leaf (text $T^k$) in $O(\log s)$ time. Computing the rank of $(1, j)$ is analogous. By comparing these two ranks, one can find the correct index in the vicinity of $\mathsf{SA}^{-1}[(1,j)]$ making a *select*() operation on the bit-vector $F$ used for *locate*() operation. The overall time is the same as for computing $\mathsf{SA}[i]$. Finally, with a gap-encoded bit vectors storing tables $C$ and $C_B$, the operation $T[SA[i]]$ works in $AT(N,\sigma)$ time. $\square$

Figure 2 illustrates the use of persistent selection for the samples.

The confluental version of Theorem 13 can be used to handle dynamic samples (see [16]).

## 6 Run-Length Compressed Suffix Tree

The entropy-bounded compressed suffix tree of Fischer et al. [7] uses an encoding of $LCP$-values (lengths of longest common prefixes of $\mathsf{SA}[i]$ and $\mathsf{SA}[i+1]$) that consists of two bit-vectors of length $N$ each containing $R$ bits set. In addition, only $o(N)$ bit structures and normal suffix array operations are used for supporting an extended set of suffix tree operations. We can now use Theorem 14 to support suffix array functionality and Theorem 9 to store $LCP$-values in $2R \log \frac{N}{R} + O\left(R \log\log \frac{N}{R}\right)$ bits. Thus, adding $2R \log \frac{N}{R} + O\left(R \log\log \frac{N}{R}\right) + o(N)$ bits to the structure of Theorem 9, one can support all the suffix tree operations listed in [7] in $O(\mathit{polylog}(N))$ time.

## 7 Discussion

We 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-

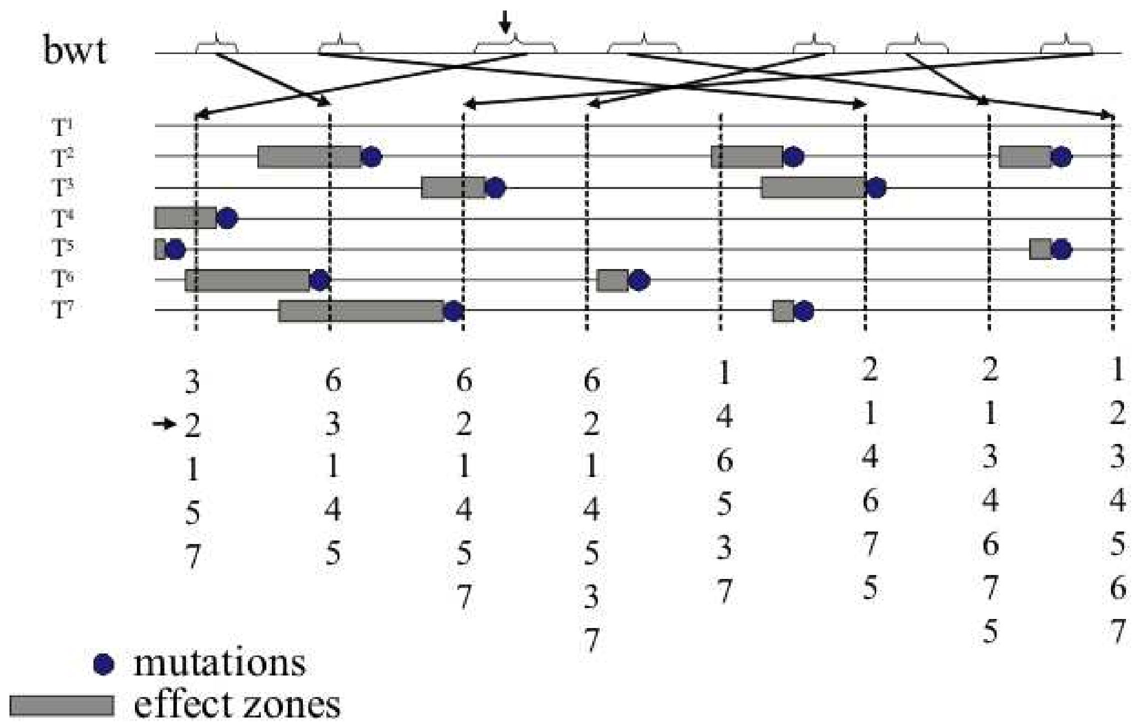


Figure 2: Persistent selection and changes in the lexicographic order of sampled suffixes. Text $T^1$ has been sampled regularly and the pointers to the sampled suffixes are stored with respect to the BWT sequence. Each such pointer is associated a range containing the occurrences of the same significant prefix in the mutated copies of $T^1$. The relative lexicographic order of these aligned suffixes (shown below the sampled positions) change only when there is a mutation effect zone between the sampled positions; when an effect zone starts, the corresponding text is removed from the list, and when it ends (with the mutation), the text is inserted to the list with a new relative lexicographic order.

Wheeler transform change only for those suffixes whose lexicographic order is affected by a mutation. In all mutation types (except in reversals) the effect to the lexicographic order of suffixes is identical to point mutations, so the expected case bounds limiting the length of significant prefixes extend easily to the real variation occurring in genomes [16]. Reverse complementation is easy to take into account as well, by adding the reverse complement of the base sequence to the collection.

The base structure (RLFM index) for counting queries is universal in the sense that it does not not need to know what and where the mutations are. This observation has been experimentally verified with throughout experimentation on version control data and on DNA sequences [26].

However, the structures for `display`() and `locate`() require the alignment of each sequence with the base sequence to be given; for succinctness we considered the easy case of identical length sequences and point mutations, where the alignment is trivial to compute. Just allowing the sequences to be of different length makes the alignment a non-trivial task. We show in the full paper [16] how to store the alignments space-efficiently

and support `display`() and `locate`() analogously to the case of point mutations covered here.

In the full paper [16], we also show how to make all the structures dynamic, solving the *dynamic repetitive collection problem*, where sequences can be inserted and deleted to and from the collection. In that case, the space bound remains the same and all time requirements are multiplied roughly by a logarithm factor. A major future challenge remains: How to remove the near linear $o(N)$ factor from compressed suffix tree space complexity?

## References


[1] D. Blanford and G. Blelloch. Compact representations of ordered sets. In *Proc. 15th SODA*, pages 11–19, 2004.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.

[3] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[4] G. M. Church. Genomes for all. *Scientific American*, 294(1):47–54, 2006.

[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] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (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 Journal 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 *DCC '06: Proceedings of the Data Compression Conference (DCC'06)*, 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] H. Kaplan. *Handbook of Data Structures and Applications (D. P. Mehta and S. Sahni Eds.)*, chapter 31: Persistent Data Structures. Chapman & Hall, 2005.

[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 and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):Article 32, 2008.

[16] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Run-length compressed indexes for repetitive sequence collections. Technical Report Technical Report C-2008-42, Department of Computer Science, University of Helsinki, Finland, 2008. http://hdl.handle.net/10138/1158.

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

[18] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

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

[20] M. H. Overmars. Searching in the past, i. Technical Report Technical Report RUU-CS-81-7, Department of Computer Science, University of Utrecht, Utrecht, Netherlands, 1981.

[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 Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 191–203, 2008.

[23] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 362–373, 2008.

[24] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal 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.

[26] J. Sirén N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. of 15th Symposium on String Processing and Information Retrieval (SPIRE 2008)*, LNCS, 2008. To appear.

[27] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, University Press, 1995.