Storage and Retrieval of Individual Genomes

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

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

Abstract. A repetitive sequence collection is one where portions of a *base sequence* of length n are repeated many times with small variations, forming a collection of total length N. Examples of such collections are version control data and genome sequences of individuals, where the differences can be expressed by lists of basic edit operations. 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 σ 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 factor remains constant when more sequences are added to the collection.
We develop a new family of self-indexes suited for the repetitive sequence collection setting. Their 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 factor is no longer constant, but depends on N/n.
We believe the structures 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.

Key words: Comparative genomics, full-text indexing, suffix tree, compressed data structures.

1 Introduction

1.1 Motivation

Self-indexing [15] is a new proposal for storing and retrieving sequence data. It aims to represent the sequence (a.k.a. text or string) compressed in a way that not only random access to the sequence is possible, but also pattern searches are supported [7, 4, 20].

⋆ Funded by the Academy of Finland under grant 119815.
⋆⋆ Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.
⋆⋆⋆ Funded by the Research Foundation of the University of Helsinki.
† Funded by the Helsinki Graduate School in Computer Science and Engineering.

A special case of a text collection is one which contains several *versions* of one or more *base sequences*. Such collections are soon becoming reality in the field of molecular biology. As the DNA sequencing technologies become faster and more cost-effective, the sequencing of individual genomes will become a feasible task [3, 10, 17]. This is likely to happen in the near future, see for example the 1000 Genomes project[3]. With such data in hand, many fundamental issues become of top concern, like how to store, say, one million Human Genomes, not to speak about analyzing them. For the analysis of such collections, one would clearly need to use some variant of a *generalized suffix tree* [9], which provides a variety of algorithmic tools to do analyses in linear or near-linear time. The memory requirement of such a solution, however, is unimaginable with current random access memories, and also challenging in permanent storage.

Self-indexes should, in principle, cope well with genome sequences, as genomes contain high amounts of repetitive structure. In particular, as the main building blocks of *compressed suffix trees* [21, 19, 18, 6], self-indexes enable compressing sequence collections close to their *high-order entropy* and enabling flexible analysis tasks to be carried out[4]. Those indexes have been successful in bringing down the space requirement of the suffix tree of *one* Human Genome to fit the capabilities of a desktop computer. However, they suffer from a fundamental limit: The high-order entropies they achieve are defined by the frequencies of symbols in their fixed-length contexts, and 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.

1.2 Content

In this paper we propose a new family of self-indexes that are suitable for storing highly repetitive collections of sequences, 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 the aligned sequences. The main technical contribution is a new strategy to store *suffix array* samples that uses and improves a classical solution for *persistent selection*.

We show analytically that the expected space requirement of our new self-indexes improves upon the existing ones on highly repetitive collections. We also provide experiments on a collection of resequenced yeast genomes, showing that our indexes behave in practice as predicted by our analysis.

1.3 Definitions and Background

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a *sequence* of *symbols* (a.k.a. characters or letters). Each symbol is an element of an *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring*

[3] `http://www.1000genomes.com`

[4] For a concrete example, the `SUDS Genome Browser` at `http://www.cs.helsinki.fi/group/suds/cst`, runs a compressed suffix tree of the Human Genome using 8.8 GB of main memory.

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 string terminated by the special symbol $t_n = \$ \notin \Sigma$, smaller than any other symbol in Σ. 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. [14]. For our purposes, it is enough to know the basic property $0 \leq H_k(T) \leq H_{k-1}(T) \leq \cdots \leq H_0(T) \leq \log \sigma$.

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 the *suffix array* [13] $\mathsf{SA}[1,n]$ 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*) denote the event of a symbol changing into another symbol inside a string. We study the following problem (other types of mutations are considered later).

Definition 1. *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for $1 \leq k \leq r$ 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 $\mathcal{C}$?); `locate`(P) (list the occurrence positions of P in $\mathcal{C}$); and `display`(k, i, j) (return $T^k_{i,j}$).*

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

A classical solution to the basic indexing problem uses T and the suffix array $\mathsf{SA}[1,n]$. Two binary searches find the interval $\mathsf{SA}[sp, ep]$ pointing to all the suffixes of T starting with P, that is, to all the occurrences of P in T (this solves `count` and `locate`) [13], and T is at hand for `display`. The solution is not space-efficient, since array SA requires $n \log n$ bits (compared to $n \log \sigma$ bits used by T), and it is not a self-index, since T is needed.

The *FM-index* [4] is a self-index based on the BWT. It solves `count` by finding the interval $\mathsf{SA}[sp, ep]$ that contains the occurrences of P. The FM-index uses the array C and function $rank_c(L, i)$ in the so-called *backward search* algorithm, calling function $rank_c(L, i)$ $O(|P|)$ times. The two other basic queries are solved using sampling of SA and its inverse SA^{-1}, and the 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 [15]. For example, on small alphabets, it is possible to achieve $nH_k + o(n \log \sigma)$ bits of space, for moderate k, with constant time support for $rank_c(L, i)$ [5].

Now, the repetitive collection indexing problem can be solved using the normal self-index for the concatenation $T^1\$T^2\$ \cdots T^r\$$. However, the space requirement achieved, even with a high-entropy compressed index, is not attractive for the case of repetitive collections. For example, an FM-index [5] requires $NH_k(C) + o(N \log \sigma)$ bits. Notice that with the collection of Def. 1 and even with $s = 0$, it holds $H_k(C) \approx H_k(T^1)$, and hence the space is about r times that for the base sequence, not taking any advantage of repetitiveness.

In the sequel, we derive solutions whose space requirements depend on n and s instead of N. Let us first consider a natural lower bound that takes into account these specific problem parameters. Consider a two-part compression scheme that compresses T^1 with a high-order compressor, and 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

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

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

Notice that it is not difficult to achieve *just plain compression* approaching the bound of Eq. (1) (omitting alphabet-dependent factors), but we aim higher: Our goal is to solve the repetitive collection indexing problem within the same space. We do not yet achieve that goal, but the space of our indexes can be expressed in similar terms; we encourage the reader to compare our final result with Eq. (1) to see the connection.

The abstract problem with point mutations studied here is much simpler than the real variations occurring in genome sequences. However, all the techniques introduced can be extended to the full set of mutation events, as is done in our implementation. This will be discussed in Sect. 3.

2 Methods

2.1 Analysis of Runs

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 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 Ψ corresponding to self-repetitions in the suffix array are called *runs*. The name stems from the

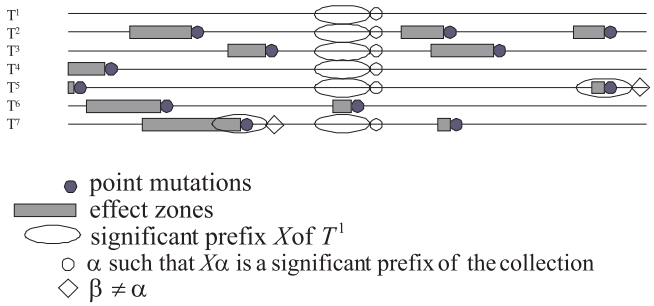

Fig. 1. An example of the significant prefix concept. Let (a repeated collection with) base text $S^1 = T^1$ contain a significant prefix X. Substring X becomes repeated in the mutated copies T^2, T^3, T^4, T^5, and T^7, of T^1. Text T^6 has a mutation inside X. Due to other mutations, texts T^5 and T^7 now contain X in some other positions, and hence X is no longer a significant prefix of the mutated collection. However, extending X with string α 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, being α the shortest extension having the required property. The significant prefixes starting at the *effect zones* shown are affected by the mutations.

fact that $\Psi(i+1) = \Psi(i) + 1$ when both $\Psi(i)$ and $\Psi(i+1)$ are contained in the same run (see [12, 15] for more details).

Let $R_\Psi(T)$ be the number of runs in Ψ of text $T = T_{1,n}$ and $R_{bwt}(T)$ the number of equal-letter runs in T^{bwt}, the BWT of T. If the text is evident from the context, we will usually drop T and write just R_Ψ and R_{bwt}. It is known that $R_\Psi \leq R_{bwt} \leq R_\Psi + \sigma$, making the two types of runs almost equal [12]. Hence, we may simplify the notation further by denoting just $R = R_{bwt}(T)$. In addition to the trivial bound $R \leq n$, we also have $R \leq nH_k + \sigma^k$ for all k [12].

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 $c \in \Sigma$. We use # as a separator between texts in the collection, and 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 $S = S_{1,n}$ is $\mathcal{S}^r = S^1 S^2 \cdots S^r$, where $S^r = S = S_{1,n-1}\$$ and $S^i = S_{1,n-1}\#$ for all $i < r$.*

Definition 3. *Let $\mathcal{T}^r$ be a collection of r texts, each derived by mutations from a base sequence $T = T^1$. The* 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 $T^k_{j,n}$, $k \neq i$.*

Notice that significant prefix concept is well-defined also a repeated collection, since it is a collection with no mutations. In that case, significant prefixes are identical to those of a collection consisting only of the base text. Figure 1 illustrates the definitions.

We now show some basic results concerning the number of runs in repeated and mutated texts. Proofs are sketched here for conciseness. Full proofs will appear in the full version. Expected case proofs extend those in [23, pp.263–265].

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

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

Lemma 2. *Let $\mathcal{S}^r = S^1 S^2 \cdots S^r$ be a repeated collection and $\mathcal{T}^r$ the collection created by transforming s^i_j, for some $1 < i \leq r$ and $1 \leq j < n$, into another character. Then $R_\Psi(\mathcal{T}^r) \leq R_\Psi(\mathcal{S}^r) + 2c + 2 = R_\Psi(S) + 2c + 2$, 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 3. *Let $S = S_{1,n}$ be a random text. The expected length of the longest repeated substring 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 a repeat of length $c \cdot \log_\sigma n$ exponentially in c^{-1}. Overlapping repeats are handled in a similar manner.

Lemma 4. *Let $\mathcal{S}^r$ be the repeated collection of random text $S = S_{1,n}$ with total length $N = nr$. Let $\mathcal{T}^r$ be $\mathcal{S}^r$ after s point mutations at random positions in $S^2 S^3 \cdots S^r$. The expected value of $R_\Psi(\mathcal{T}^r)$ is at most $R_\Psi(S) + O(s \log_\sigma N)$.*

The above analysis can be extended to other types of mutations. When we insert a new copy of an existing substring, the significant prefixes completely within the new copy remain unchanged. Only the significant prefixes covering either end of the inserted copy change. Hence the insertion is essentially equivalent to two point mutations. Similarly the deletion of a substring is equivalent to one point mutation.

2.2 Backward Search for Repetitive Collections

Our prior work [22] introduced three solutions to the repetitive collection indexing problem, restricted to $\texttt{count}(P)$ query. The three indexes are *Run-Length Compressed Suffix Array (RLCSA)*, *Run-Length Encoded Wavelet Tree (RLWT)*, and *Improved Run-Length FM-Index (RLFM+)*. They achieve different space vs. time trade-offs. For example, RLFM+ requires space $(R \log \sigma + 2R \log \frac{N}{R})(1 + o(1)) + O\left(R \log\log \frac{N}{R}\right)$ bits. Query $rank_c(T^{bwt}, i)$, and retrieving symbol t^{bwt}_i, are solved in $O(t_{\mathsf{LF}})$ time, where $t_{\mathsf{LF}} = O(\log R)$.[5] Hence, RLFM+ supports

[5] One can achieve $o((\log\log N)^2)$ time by adding $O(R\frac{\log N}{\log R})$ further bits of space [8].

`count`(P) in time $O(|P| t_{\mathsf{LF}})$. Lemma 4 can be used to bound R with $n + O(s \log_\sigma N)$ in the expected case (even for an incompressible T^1), which gives a space bound close to the terms of Eq. (1).

2.3 Suffix Array Samples

Supporting the other two functions of the repetitive collection indexing problem, namely, `display`() and `locate`(), is the main contribution of this paper. We address this now.

We need to be able to map the suffixes of the text into suffix array indexes and vice versa. The standard solution [15] 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 $D[i] = \mathsf{SA}^{-1}(i \cdot d)$, mark the locations $D[i]$ in 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 $S[1, N/d + 1]$, such that $S[rank_1(B, D[i])] = i \cdot d$.

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

Function `locate(P)` works as follows. First, backward search finds the range $\mathsf{SA}[sp, ep]$ containing the occurrences of P, and then 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 $t_{\mathsf{SA}} = O(t_{\mathsf{LF}} \cdot d)$.

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 the *binary searchable dictionary (BSD)* representation [8]; this changes the time for `locate`() into $t_{\mathsf{SA}} = O((t_{\mathsf{LF}} + t_{\mathsf{SA}})d)$, where $t_{\mathsf{SA}} = O(\log d)$.

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.

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 $T^1_{i \cdot d, n}$ stored at $D[i] = \mathsf{SA}^{-1}[i \cdot d]$.

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 `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 BSD [8] 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 $O((n/d) \log n)$.

Improving Space for `locate()` We use the same strategy as for `display`(), sampling only T^1, 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 SA. Assume every d-th suffix of T^1 is sampled and those sampled SA positions are marked in a bit vector B. Then 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 $j - j' \leq r$. The candidate $j' < j$ to check is $j' = select_1(B, rank_1(B, j))$, where $select_1(B, x)$ gives the position of the x-th 1 in B. 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 SA, $c < d$ is the number of times LF-mapping was applied, and $k = j - j' + 1$.

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 2; 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 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 $Q[rank_1(B, \mathsf{SA}^{-1}[i \cdot d])] = k_1 k_2 \cdots k_p$ denoting texts $T^{k_1}, T^{k_2}, \ldots, T^{k_p}$, $p \leq r$, that correspond to a sampled suffix $T^1_{i \cdot d,n}$. 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 close enough position $\mathsf{SA}[j']$.

Let us consider subproblem (ii) first. A solution is to explicitly mark all class B suffixes in SA into a bit-vector $F[1, N]$, and to store for each sampled suffix $T^1_{i \cdot d,n}$ its lexicographic *rank* e among the suffixes in the list $Q[rank_1(B, \mathsf{SA}^{-1}[i \cdot d])] = k_1 k_2 \cdots k_p$, 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 $Q[rank_1(B, prev)] = k_1k_2 \cdots k_p$ 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 $Q[rank_1(B, succ)] = k_1k_2 \cdots k_{p'}$ 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 [16] to *k^{th} element/rank searching in the past*. The original structure is reviewed in Theorem 1 and then Theorem 2 improves the space and makes the structure *confluental persistent* (see [11] for background).

Definition 4. *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} (or before e_1^{tprev} or after $e_{p_t}^{tprev}$). 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 1 ([16]). *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 persistence: 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 [16] 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 [16]. 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 2. *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, where x is the number of insertions and deletions to construct $\mathcal{D}$. 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 1 by replacing the time-stamped lists of fields in each node of the tree with two partial sums that can be represented succinctly. Let $S^v = s_0^v s_1^v s_2^v, \cdots s_{k^v}^v$ be the list of subtree sizes stored in some node v, where $s_0^v = 0$. Let $\hat{S}^v = (s_1^v - s_0^v)(s_2^v - s_1^v) \cdots (s_{k^v}^v - s_{k^v-1}^v)$. We represent $\hat{S}^v$ via succinct data structure for (dynamic) partial sums to support operations $select(\hat{S}^v, i) = \sum_{j=1}^{i} \hat{s}_j^v = s_i^v$. In addition, we construct a bit-vector $B^v[1, k^v]$ where $B^v[i] = 1$ if and only if the change s_i^v 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 $i' = rank_1(B^v, i)$, 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_i^v. 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 $select(\hat{S}^v, i) = s_i^v$. First we observe that $\sum_{v \in \mathcal{T}} \sum_{j=1}^{k^v} \hat{s}_j^v = O(x \log x)$ 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 $F^v = f(\hat{s}_1^v) f(\hat{s}_2^v) \cdots f(\hat{s}_{k^v}^v)$, where $f(x) = 1^x$ if $x > 0$ otherwise $f(x) = 0^{-x}$, and by a bit-vector $G = 10^{|f(\hat{s}_1^v)|-1} 10^{|f(\hat{s}_2^v)|-1} \cdots 10^{|f(\hat{s}_{k^v}^v)|-1}$. 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 Lemma 4 and RLFM+ structure of Sect. 2.2 with Theorem 2 applied to sampling gives us the main result of the paper:

Theorem 3. *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 indexing problem *taking*

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

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

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 2 provides a solution to 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, see Fig. 2 for an example). 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 $O(s \log s)$ bits. Computation of $\mathsf{SA}[i]$ is identical to *locate*(), but computation of $\mathsf{SA}^{-1}[(k, j)]$ needs some interplay with the structures of Theorem 2, considered next.

In the case t_j^k belongs to an area where a sampled position is at distance d, computation of $\mathsf{SA}^{-1}[(k, j)]$ resembles the display operation. Otherwise one must follow the 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 2 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

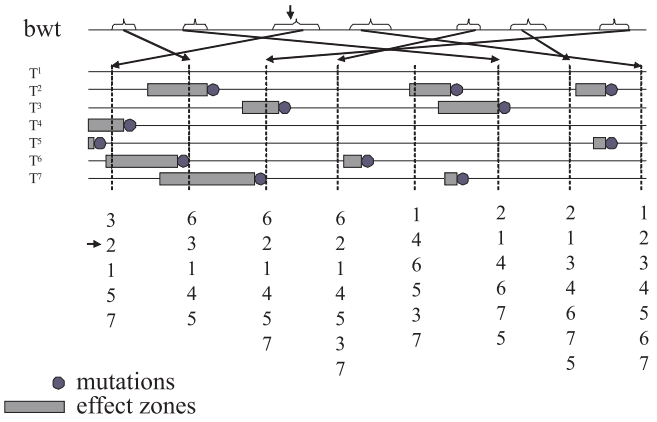

Fig. 2. Persistent selection and changes in the lexicographic order of sampled suffixes. Text T^1 has been sampled regularly and the pointers to the sampled suffixes are stored with respect to the BWT sequence. Each such pointer is associated a range containing the occurrences of the same significant 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.

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_{SA} 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.

The confluental version of Theorem 2 can be used to handle dynamic samples.

The result can be used to derive new compressed suffix trees: The entropy-bounded compressed suffix tree of Fischer et al. [6] 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 3 to support suffix array functionality and BSD representation [8] to store LCP-values in $2R \log \frac{N}{R} + O\left(R \log \log \frac{N}{R}\right))$ bits. Thus, adding $2R \log \frac{N}{R} + O\left(R \log \log \frac{N}{R}\right)) + o(N)$

Table 1. Base structure sizes and times for `count`() and `display`() for various self-indexes on a collection of genomes of multiple strains of Saccharomyces paradoxus (36 sequences, 409 MB). The genomes were obtained from the Durbin Research Group at the Sanger Institute (`http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp/`). Ψ sampling rate was set to 128 in CSA and to 32 bytes in RLCSA. Reported times are in microseconds / character.

Index	Size (MB)	`count`()	`display`()
CSA	95.51	2.86	0.41
SSA	121.70	0.48	0.40
RLFM	146.40	1.21	1.38
RLCSA	42.78	1.93	0.77
RLWT	34.67	17.30	10.24
RLFM+	54.77	3.03	2.10

bits to the structure of Theorem 3, one can support all the suffix tree operations listed in [6] in $O(polylog(N))$ time.[6]

3 Implementation and Experiments

So far we have considered only point mutations on DNA, although there are many other types of mutations, like insertions, deletions, translocations, and reversals. The runs in the Burrows-Wheeler transform change only for those suffixes whose lexicographic order is affected by a mutation. In all mutation types (except in reversals) the effect to the lexicographic order of suffixes is similar to point mutations, so the expected case bounds limiting the length of significant prefixes extend easily to the real variation occurring in genomes. Reverse complementation is easy to take into account as well, by adding the reverse complement of the base sequence to the collection.

The base structures (e.g. RLFM+ index) for counting queries are universal in the sense that they do not need to know what and where the mutations are. Standard sampling techniques can be used to add reasonably efficient support for locating and displaying, as shown in Table 1 and Fig. 3. Compressed Suffix Array (CSA) [20], Succinct Suffix Array (SSA) [12, 5], and Run-Length FM-index (RLFM) [12] are existing indexes similar to our RLCSA, RLWT, and RLFM+, respectively.

The experiments were performed on a 2.66 GHz Intel Core 2 Duo system with 3 GB of RAM running Fedora Core 8 based Linux. Counting and locating times are averages over 1000 patterns of length 10. Displaying a substring of length l requires the extraction of $d/2 + l$ characters on the average.

[6] We remark that the solution is not quite satisfactory, as in $o(N)$ space one can afford to use the standard suffix array sampling as well. Converting $o(N)$ to $o(n)$ is an open problem, and it seems to be common to all different compressed suffix tree approaches.

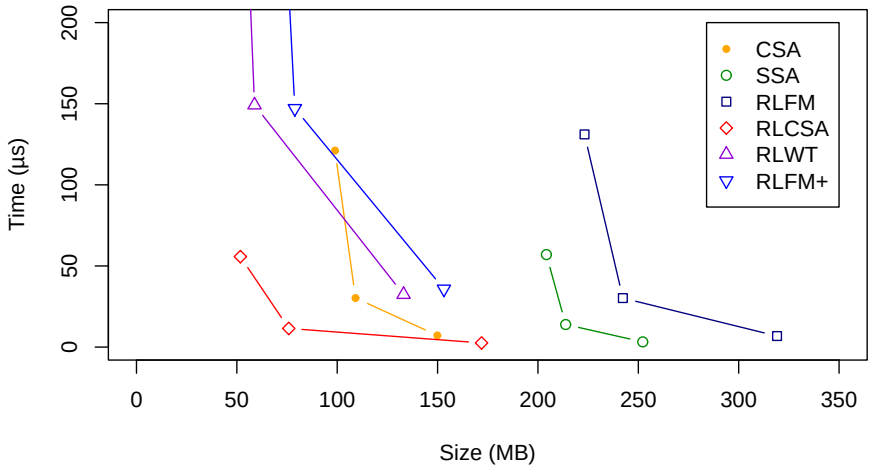

Fig. 3. Sizes and times for `locate`() for self-indexes on the S paradoxus collection. Each index was tested with sampling rates $d = 32$, 128, and 512. Reported times are in microseconds / occurrence.

RLWT and RLFM+ currently store the suffix array samples in a more space efficient manner than RLCSA. By using the same method for all three indexes, the size differences between them would be determined only by the sizes of base structures for counting. On the other hand, `locate`() in RLCSA and RLWT has been optimized for retrieving multiple occurrences in parallel. As the base structures are run-length encoded, we can perform the base step (LF-mapping in RLWT) for an entire run for roughly the same cost as for a single occurrence. If similar optimizations were implemented in RLFM+, its locating speed would be close to that of RLCSA.

The new 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). The structures are easy to extend to more general alignments. Our current implementation supports alignments with gaps (i.e. runs of `N`s) as well as insertions and deletions in addition to substitutions.

The main component required is the static structure supporting persistent selection. For its construction, we implemented also the dynamic structure supporting online persistent selection (with minor modifications it would support confluental persistent selection as well). Once it is constructed for the given alignment, it is converted into a static structure. The static structure is in fact more space-efficient than the one described in Theorem 2, as we discard

Table 2. Standard sampling versus persistent selection. The rows give the size of the base structure (RLWT), size of suffix array samples, size of display structures, size of persistent selection structure including bookkeeping of zones, and the total size.

	Mutation rate 0.001		Mutation rate 0.0001	
Approach/Size	Standard	Persistent	Standard	Persistent
Base (MB)	4.06	4.06	2.19	2.19
Samples (MB)	1.24	0.28	1.02	0.25
Display (MB)		0.32		0.07
Persistent (MB)		3.22		0.31
Total size (MB)	5.30	7.89	3.21	2.82

completely the tree structure and instead concatenate levelwise the two bitvectors stored at the nodes of the tree; a third bitvector is added marking the leaves, which enables us to navigate in the tree whose nodes are now represented as ranges. The time-to-rank mapping in the root of the persistent tree can be stored space-efficiently using the BSD representation. The space requirement is $6x \log x(1 + o(1)) + x \log \frac{h}{x}(1 + o(1)) + O(x \log \frac{h}{x}) + x \log r$ bits, where $6x \log x(1 + o(1))$ comes from the 3 bitvectors of length x supporting *rank* and *select* on each of the at most $2 \log x$ levels of the red-black balanced tree, $x \log \frac{h}{x}(1 + o(1)) + O(x \log \frac{h}{x})$ comes from the BSD representation, and $x \log r$ from the values stored at leaves.

The interesting question is at which mutation rates the persistent selection approach will become competitive with the standard sampling approach. With the mutation rates occurring in yeast collection of Fig. 3, the persistent selection approach does not seem to be a good choice; it occupied 8.49 MB on the 36 strains of S paradoxus chromosome 2 (28.67 MB), while RLWT with standard sampling approach occupied 3.97 MB.[7] The sampling parameters were chosen so that both approaches obtained similar time efficiency (403 versus 320 microseconds for one locate, respectively). To empirically explore the turning point where the persistent selection approach becomes competitive, we generated a DNA sequence collection with 100 copies of a 1 MB reference sequence and applied different amount of random mutations on it. Table 2 illustrates the turning point by giving the space requirements for RLWT+sampling versus RLWT+persistent selection on two different mutation rates, where their order changes. We used sampling rate $d = 512$ for standard sampling, and $d = 64$ ($d = 32$) for persistent selection approach on mutation rate 0.001 (on mutation rate 0.0001). This made the running times reasonably close; for example, one locate took 172 versus 184 microseconds on 0.0001 mutation rate, respectively.

[7] We observed that the given multiple alignments were not the best possible; the size would be reduced significantly by the choice of better consensus sequences.

4 Conclusions

We have studied the problem of representing highly repetitive sequences in such a way that their repetitiveness is exploited to achieve little space, yet at the same time any part of the sequences can be extracted and searched without decompressing it. This problem is becoming crucial in Computational Biology, due to the cheaper and cheaper availability of sequence data and the interest in analyzing it.

We have shown that the current compressed text indexing technology is not well suited to cope with this problem, and have devised variants that have shown to be much more successful.

In the full paper we will show how to allow for dynamism, that is, permitting to handle a compressed collection where insertion and deletion of sequences can be efficiently intermixed with searches. We achieve the same space bounds, whereas the time requirements are multiplied roughly by a logarithmic factor.

An important challenge for future work is to look for schemes achieving further compression. For example, LZ77 algorithm is an excellent candidate to compress repetitive collections, achieving space proportional to the number of mutations. For example, the 409 MB collection of Saccharomyces paradoxus strains studied here can be compressed into 4.93 MB using an efficient LZ77 implementation[8]. This is over 7 times less space than what the new self-indexes studied in this paper achieve. Yet, LZ77 has defied for years its adaptation to a self-index form. Thus there is a wide margin of opportunity for such a development.

Acknowledgments

We wish to thank Teemu Kivioja from Institute of Biomedicine, University of Helsinki, for turning our attention to the challenges of individual genomes. We wish also to thank Kimmo Palin from Sanger Institute, Hinxton, for pointing us the yeast genome collection.

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. G. M. Church. Genomes for all. *Scientific American*, 294(1):47–54, 2006.
4. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
5. 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.

[8] http://p7zip.sourceforge.net/

6. 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.
7. 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.
8. 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.
9. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
10. N. Hall. Advanced sequencing technologies and their wider impact in microbiology. *The Journal of Experimental Biology*, 209:1518–1525, 2007.
11. H. Kaplan. *Handbook of Data Structures and Applications (D. P. Mehta and S. Sahni Eds.)*, chapter 31: Persistent Data Structures. Chapman & Hall, 2005.
12. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
13. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
14. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
15. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
16. 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.
17. E. Pennisi. Breakthrough of the year: Human genetic variation. *Science*, 21:1842–1843, December 2007.
18. 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.
19. 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.
20. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
21. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
22. 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 5280, pages 164–175, 2008.
23. M. S. Waterman. *Introduction to Computational Biology.* Chapman & Hall, University Press, 1995.