JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 17, Number 3, 2010

Pp. 281–308
DOI: 10.1089/cmb.2009.0169

# Storage and Retrieval of Highly Repetitive Sequence Collections

VELI MÄKINEN,[1] GONZALO NAVARRO,[2] JOUNI SIRÉN,[1] and NIKO VÄLIMÄKI[1]

## ABSTRACT

**A repetitive sequence collection is a set of sequences which are small variations of each other. A prominent example are genome sequences of individuals of the same or close species, where the differences can be expressed by short lists of basic edit operations. Flexible and efficient data analysis on such a typically huge collection is plausible using suffix trees. However, the suffix tree occupies much space, which very soon inhibits in-memory analyses. Recent advances in full-text indexing reduce the space of the suffix tree to, essentially, that of the *compressed* sequences, while retaining its functionality with only a polylogarithmic slowdown. However, the underlying compression model considers only the predictability of the next sequence symbol given the $k$ previous ones, where $k$ is a small integer. This is unable to capture longer-term repetitiveness. For example, $r$ identical copies of an incompressible sequence will be incompressible under this model. We develop new static and dynamic full-text indexes that are able of capturing the fact that a collection is highly repetitive, and require space basically proportional to the length of one typical sequence plus the total number of edit operations. The new indexes can be plugged into a recent dynamic fully-compressed suffix tree, achieving full functionality for sequence analysis, while retaining the reduced space and the polylogarithmic slowdown. Our experimental results confirm the practicality of our proposal.**

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

## 1. INTRODUCTION

### 1.1. Motivation

*SELF-INDEXING* (Navarro and Mäkinen, 2007) is a new proposal for storing and retrieving sequence data. It aims to represent the sequence (a.k.a. text or string) compressed in such a way that not only random access to the sequence is possible, but also efficient pattern searches are supported (Grossi and Vitter, 2006; Ferragina and Manzini, 2005; Sadakane, 2003).

The self-indexing approach becomes especially interesting when applied to collections of texts. Consider for example a file system that is automatically kept self-indexed. Files can be accessed without decompressing, and searched by content in real time as queries use the index rather than scanning through the

[1]Department of Computer Science, University of Helsinki, University of Helsinki, Helsinki, Finland.
[2]Department of Computer Science, University of Chile, Santiago, Chile.

files. Such retrieval functionalities have been available for long time on natural language texts by the well-known *inverted indexes*, but now self-indexes make such retrieval possible, within reasonable memory space, for arbitrary texts such as biological sequences that do not consist of separable words.

A special case of a text collection is one which contains several *versions* of one or more *base sequences*. Such collections are not uncommon. For example, a *version control system* (e.g., for collaborative document editing or software development) needs to store several versions of the same file with only small edit differences between the consecutive entries. If the entries are stored independently of each other, the version control system will end up spending unnecessarily large amounts of memory. If the system stores only the edits, queries on the content of one specific version becomes non-trivial.

An analogy to the storage and retrieval of version control data is soon becoming reality also 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 (Church, 2006; Hall, 2007; Pennisi, 2007). This is likely to happen in the near future, see for example the 1000 Genomes project.[1] With such data in hand, many fundamental issues become of top concern, like how to store, say, 1000 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* (Gusfield, 1997), 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* (Sadakane, 2007; Russo et al., 2008a,b; Fischer et al., 2008), self-indexes enable compressing sequence collections close to their *high-order entropy* while enabling flexible analysis tasks to be carried out.[2]

A very successful direction in the practical use of self-indexes for genome sequences has been the *short read mapping* problem that stems from the new high-throughput sequencing technologies; instead of trying to assemble a genome from sequencing reads, the goal is just to map a set of short fragments to their best occurrences in the reference genome. The short reads are specific to the experimental setup and enable analyzing for example alternative splicing or individual genetic variation. Mapping the reads to the genome can be done efficiently using suffix tree (or compressed suffix tree), but the *backtracking* functionality required for the task is already supported by the underlying base self-index (Lam et al., 2008). This observation has led to several carefully tailored practical tools for the short read mapping problem (Langmead et al., 2009; Li and Durbin, 2009).

The above approaches have been successful in bringing down the space requirement of a powerful index structure for *one* Human Genome to fit the capabilities of a desktop computer. However, even the most compressed self-indexes up-to-date 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, and therefore do not scale up to solve the problem of managing repetitive sequence collections.

### 1.2. Content

In this article, 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. This is an extension of the classical objective of DNA compression in the *vertical mode* (Grümbach and Tahi, 1993, 1994; Giancarlo et al., 2009), namely, where the goal is to compress a collection of genomes such that each sequence is compressed by making use of information contained in the entire set.

Our main technical contributions are (1) adaptations of existing self-indexes so that their compression performance goes beyond the high-order entropy model; (2) a new strategy to store *suffix array* samples

[1] www.1000genomes.com.

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

that uses and improves a classical solution for *persistent selection*; (3) an analytical proof that the expected space requirement of our new self-indexes improves upon the existing ones on highly repetitive collections; and (4) a proof-of-concept implementation that demonstrates the practicality of our results. We provide experiments on a collection of resequenced yeast genomes showing that our indexes behave in practice as predicted by our analysis.

The article is structured as follows. Section 1.3 introduces the basic concepts and defines formally the problems to be studied. Section 2 contains the structural analyses showing where compression can be achieved in the context of repetitive collections. Section 3 introduces three different base structures that exploit the analyzed structural properties to achieve compression. Section 4 adds the standard suffix array sampling scheme on top of the new base structures to give a practical solution to the storage and retrieval of repetitive collections. Section 5 develops an advanced suffix array sampling scheme to achieve a better theoretical solution. Section 6 discusses how the new structures can be used as backbones of recent fully-compressed suffix trees. Section 7 describes the implementations and shows that the new structures achieve good space bounds on real data sets. Section 8 concludes by discussing the extensions and possible directions for future research.

### 1.3. Definitions and background

*1.3.1. Basic notions.* A *string* $S = S_{1,n} = S[1,n] = S[1]S[2]\ldots S[n] = s_1 s_2 \ldots s_n$ 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* of $S$ is written $s_{i,j} = s[i, j] = s_i s_{i+1} \ldots s_j$. 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 $\Sigma$. The symbol "$<$" among strings denotes the classical *lexicographical order*.

We use the standard notion of *empirical k-th order entropy* $H_k(T)$ (Cover and Thomas, 1991; Manzini, 2001). The zero-order entropy is defined as $H_0(T) = \Sigma_{1 \le c \le \sigma} \frac{n_c}{n} \log \frac{n}{n_c}$, where $n_c$ is the number of occurrences in $T$ of character $c$. (We write log for $\log_2$ throughout this article.) For $k > 0$ we define $H_k(T) = \frac{1}{n} \Sigma_{S \in \Sigma^k} |T_S| H_0(T_S)$, where $T_S$ is a string formed by the characters following the occurrences of substring $S$ in $T$. It holds $0 \le H_k(T) \le H_{k-1}(T) \le \ldots \le H_0(T) \le \log \sigma$.

*1.3.2. Suffix array and Burrows-Wheeler transform.* The *suffix array* (Manber and Myers, 1993) SA[1,$n$] of a text $T$[1,$n$] is a permutation of the positions $\{1, \ldots, n\}$ such that the suffixes $T[\mathrm{SA}[i], n]$ are listed in lexicographic order as $i$ increases. Because every substring of $T$ is a prefix of a suffix of $T$, and the suffixes prefixed by a string $P$ form a lexicographic range in SA, it turns out that the starting positions of all the occurrences of a string $P$ in $T$ are found within an interval SA[$sp$,$ep$], which can be found by binary search.

The compressors to be discussed are derivatives of the *Burrows-Wheeler transform (BWT)* (Burrows and Wheeler, 1994). The transform produces a permutation of $T$, denoted by $T^{bwt}$, as follows: (i) build the suffix array SA of $T$; (ii) the transformed text is $T^{bwt} = L$, where $L[i] = T[\mathrm{SA}[i] - 1]$, 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 $i \in n-1, \ldots, 1$ do $t_i \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally, put the end marker $t_n \leftarrow \$$.

We study a generalization of the following problem:

**Definition 1.** *The* basic (compressed) indexing problem *is to store text T in as small space as possible, so that the following retrieval queries on any given* pattern *string* $P = p_1 p_2 \ldots p_m$ *can be solved as efficiently as possible:*

count($P$): *How many times P appears as a substring of T?*
locate($P$): *List the occurrence positions of P in T.*
display($i$,$j$): *Return* $T_{i,j}$.

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

A comprehensive solution to the basic indexing problem uses the suffix array SA$[1, n]$. Then two binary searches are enough to find the interval SA[*sp, ep*] such that count and locate are immediately solved (Manber and Myers, 1993). The solution is not as space-efficient as possible, since array SA requires $n$ log $n$ bits, and the solution is not yet a self-index, since $T$ is needed in order to solve the display query.

*1.3.3. Self-indexing.* A more space-efficient solution to the basic indexing problem exploits the connection of SA and BWT: The *FM-index* (Ferragina and Manzini, 2005) is a self-index that solves counting queries by finding the interval SA[*sp, ep*] just based on the BWT. The solution 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. Its pseudocode is given next:

**Algorithm** Count ($P[1 \ldots m], L[1 \ldots n]$)

(1) $i \leftarrow m$;
(2) $sp \leftarrow 1$; $ep \leftarrow n$;
(3) **while** $(sp \leq ep)$ **and** $(i \geq 1)$ **do**
(4) $s \leftarrow P[i]$;
(5) $sp \leftarrow C[s] + rank_s(L, sp - 1) + 1$;
(6) $ep \leftarrow C[s] + rank_s(L, ep)$;
(7) $i \leftarrow i - 1$;
(8) **if** $(ep < sp)$ **then return** "not found"
**else return** "found $(ep - sp + 1)$ occurrences".

The correctness of the above algorithm is easy to see by induction: At each phase $i$, [*sp,ep*] gives the maximal interval of suffix array SA pointing to suffixes prefixed by $P$ [*i,m*].

The two other basic indexing problem queries are solved, e.g., using a sampling of SA and its inverse, and *LF*-mapping to derive the unsampled values from the sampled ones. Many variants of the FM-index have been derived that differ mainly in the way the $rank_c(L,i)$-queries are solved (Navarro and Mäkinen, 2007). For example, on small alphabet sizes, it is possible to achieve $nH_k + o(n)$ bits with constant time support for $rank_c(L,i)$ (Ferragina et al., 2007).

The backward search algorithm above can be extended to *backward backtracking* (Lam et al., 2008). The idea is to alter the backward search to branch recursively to different ranges $[sp', ep']$ representing the suffixes of the text prefixes (i.e. substrings). This is done simply by computing $sp'_c = C[c] + rank_c(L, sp - 1) + 1$ and $ep'_c = C[c] + rank_c(L, ep)$ for all $c \in \Sigma$ at each step and recursing on each [ $sp'_c$, $ep'_c$]. Then the pattern can be compared against all substrings of the text, allowing to search for approximate occurrences (Lam et al., 2008). The running time becomes exponential in the number of errors allowed, but different branch-and-bound techniques can be used to obtain practical running times (Langmead et al., 2009; Li and Durbin, 2009). Hence, any FM-index variant can in practice be used to solve slightly more complex search problems than the basic indexing problem.

A dual approach to solving the basic indexing problem uses the *compressed suffix array (CSA)* (Sadakane, 2003), which is a self-index based on an earlier succinct data structure (Grossi and Vitter, 2006). In the CSA, the suffix array SA[1, $n$] is represented by a sequence of numbers $\psi(i)$, such that $SA[\psi(i)] = SA[i] + 1$.[3] The sequence $\psi$ is differentially encoded, $\psi\ (i + 1) - \psi(i)$. Note that the $\psi$ values are increasing in the areas of SA where the suffixes start with the same character $c$, because $cX < cY$ if and only if $X < Y$ in lexicographic order. It is enough to store those increasing values differentially with a method like Elias coding to achieve $O(nH_0)$ overall space (Sadakane, 2003). Some additional information is stored to permit constant time access to $\psi$. This includes the same $C$ array used by the FM-index.

*1.3.4. Repetitive collections.* Let a *point mutation* denote the event of a symbol changing into another symbol inside a string.

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

[3]Since $SA[1] = n$ because $T[n,n] = \$$ is the smallest suffix, it should hold $SA[\psi(1)] = n + 1$. For technical convenience we set $\psi(1)$ so that $SA[\psi(1)] = 1$, which makes $\psi$ a permutation of $[1, n]$.

**Definition 2.** *Given a collection $\mathcal{C}$ of $r$ sequences $T^k \in \mathcal{C}$ such that $|T^\kappa| = n$ for each $1 \leq k \leq r$ and $\sum_{k=1}^{r} |T^k| = N$, where $T^2, T^3, \ldots, T^r$ contain overall $s$ point mutations from the base sequence $T^1$, the* repetitive collection indexing problem *is to store $\mathcal{C}$ in as small space as possible such that the following operations are supported as efficiently as possible:*

count($P$): *How many times $P$ appears as a substring of the texts in $\mathcal{C}$?*
locate($P$): *List the occurrence positions of $P$ in $\mathcal{C}$, i.e., pairs $(k,i)$ such that $\rightarrow T^\kappa[i, i + |P| - 1] = P$.*
display($k,i,j$): *Return $T^k_{i,j}$.*

We also study an extended version of the problem, where the sequences do not need to be of the same length, and the differences can also be insertions and deletions in addition to point mutations.

**Definition 3.** *Let $\mathcal{C}$ be a collection of $r$ sequences $T^k \in \mathcal{C}$ such that $|T^1| = n$, $\sum_{k=1}^{r} |T^k| = N$ and*

$$\sum_{i=2}^{r} \min_{1 \leq \alpha_i \leq \beta_i \leq |T^1|} \{d_L(T^i, T^1_{\alpha_i, \beta_i})\} = s,$$

*where $d_L(T,T')$ is the Levenshtein distance (Levenshtein, 1966) between strings $T$ and $T'$. The* repetitive substring collection indexing problem *is to store $\mathcal{C}$ in as small space as possible such that the operations count($P$), locate($P$), and display($k,i,j$) of Def. 2 are supported as efficiently as possible.*

These collection indexing problems can be solved easily using the normal self-indexes for the concatenation $\mathcal{T} = T^1 \# T^2 \# \ldots T^r\$$, where # is a special symbol not appearing in $\Sigma$.

However, the space requirements achieved even with the high-entropy compressed self-indexes are not attractive for the case of repetitive collections. For example, the solution by Ferragina et al. (2007) requires $NH_k(\mathcal{T}) + o(N \log \sigma)$ bits. Notice that with the collection of Def. 2 and with $s = 0, H_k(\mathcal{T}) \approx H_k(T^1)$, and hence the space is about $r$ times more than what the same solution uses for the basic indexing problem.

In the sequel, we derive solutions whose space requirements depend 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 edit operations needed to convert each other sequence into a substring of $T^1$. A lower-bound for any such compressor is

$$nH_k(T^1) + 2(r-1)\log n + \log \binom{N-n+s}{s} + s \log(2\sigma), \tag{1}$$

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 which substrings of $T^1$ the $r-1$ other sequences correspond to, the third part is the lower bound for telling the positions of the edit operations among the $N-n$ possible (with repetitions, for insertions), and the fourth part is the lower bound for listing the $s$ edit operations ($\sigma - 1$ possible mutations, $\sigma$ possible insertions, 1 deletion).

Notice that it is not difficult to achieve *just plain compression* approaching the bound of Eq. (1), but we aim higher: Our goal is to solve the repetitive collection indexing problems 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.

We also show how to apply the collection indexes as building blocks to turn the new *(dynamic) fully-compressed suffix trees* (Russo et al., 2008a,b; Fischer et al., 2008) into a space expressed in the framework of the lower bound of Eq. (1).

The abstract problem with point mutations and indels studied here is much simpler than the real variations occurring in the case of genome sequences. However, we emphasize that the chosen model is sufficient in representing any collection in question, as the mutations spanning larger regions (translocations, reversals) can always be represented as runs of insertions or deletions. In this case, $s$ is just much larger than if the global operations were taken into account in the data structures directly. Proper extensions to the complete set of mutations are discussed in the analyses when applicable and summarized in Section 8.

## 2. ANALYSIS OF RUNS

### 2.1. Combinatorial properties

*Self-repetitions* are the fundamental source of redundancy in suffix arrays, enabling their compression. A self-repetition is a maximal interval $SA[i,i+l]$ of suffix array SA having a *target interval* $SA[j,j+l]$ such that $SA[j+r] = SA[i+r]+1$ for all $0 \le r \le l$. Let $\psi(i) = SA^{-1}[SA[i]+1]$. The intervals of $\psi$ corresponding to self-repetitions in the suffix array are called *runs*. The name stems from the fact that $\psi(i+1) = \psi(i)+1$ when both $\psi(i)$ and $\psi(i+1)$ are contained in the same run (Mäkinen and Navarro, 2005; Navarro and Mäkinen, 2007).

Let $R_\psi(T)$ be the number of runs in $\psi$ for text $T[1,n]$ and $R_{bwt}(T)$ the number of equal-letter runs in $T^{bwt}$, the BWT of $T$. If $T$ is evident from the context we will write just $R_\psi$ and $R_{bwt}$. The two types of runs are almost equal. If we restrict the definition of self-repetitions to allow only suffixes starting with the same character, then $R_{bwt}$ is also the number of runs in $\psi$, and hence $R_\psi \le R_{bwt} \le R_\psi + \sigma$ (Mäkinen and Navarro, 2005). Thus we can simplify the notation further by denoting just $R = R_{bwt}(T)$.

In addition to the trivial bound $R \le n$, we also have $R \le nH_k + \sigma^k$ for all $k$ (Mäkinen and Navarro, 2005). We will now prove some further bounds for texts obtained by repeating and mutating substrings of a base sequence. We will make use of the character # we have introduced to separate texts in the collection, assuming $\# < \$ < c$ for all $c \in \Sigma$. We also 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. Hence, we never have to continue a comparison between two suffixes after the first # or \$ encountered.

**Definition 4.** *The r-times* repeated collection *of base text* $S = S_{1,n}$ *is* $\mathcal{S}^r = S^1 \ldots S^r$, *where* $S^r = S = S_{1,n-1}\$$ *and* $S^i = S_{1,n-1}\#$ *for all* $i < r$.

**Definition 5.** *Let* $\mathcal{T}^r$ *be a collection of r texts, each derived by mutations from a base sequence* $T = T^1$. *The* significant prefix $SP_{i,j}$ *is the shortest prefix of* $T^i_{j,n}$ *not occurring anywhere else in* $\mathcal{T}^r$ *as a substring* except *possibly as a prefix of some* $T^k_{j,n}$, $k \ne i$.

Notice that the significant prefix concept is well-defined also in a repeated collection, i.e., a collection with no mutations. In that case, significant prefixes are identical to those of a collection consisting only of the base text. Note also that the significant prefix of a suffix defines its position in the suffix array. Figure 1 illustrates the definitions.

**Lemma 6.** *For all texts S and all* $r \ge 1$, $R(S) = R(\mathcal{S}^r)$.

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

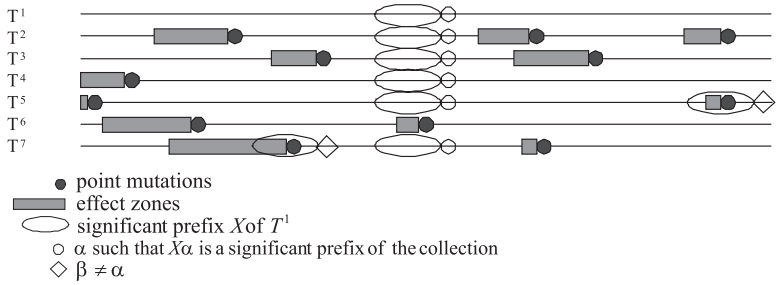


**FIG. 1.** An example of the significant prefix concept. Let (a repeated collection with) base text $S^1 = T^1$ contain a significant prefix $X$. Substring $X$ becomes repeated in the mutated copies $T^2$,$T^3$,$T^4$,$T^5$, and $T^7$, of $T^1$. Text $T^6$ has a mutation inside $X$. Due to other mutations, 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 $\alpha$ makes $X\alpha$ unique to the original position of $X$, while the other two occurrences of $X$ are succeeded by string $\beta \ne \alpha$. Hence, $X\alpha$ is a significant prefix, being $\alpha$ the shortest extension having the required property. The significant prefixes starting at the *effect zones* shown are affected by the mutations.

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

By the definition of self-repetitions, SA[$i$] and SA[$i+1$] are contained in the same self-repetition of SA if and only if the range SA$^r$[$r(i-1)+1, r(i+1)$] is contained entirely in a self-repetition of SA$^r$. Hence there is one-to-one correspondence between the self-repetitions of SA and SA$^r$. ■

**Lemma 7.** *Let $\mathcal{S}^r = S^1S^2 \ldots S^r$ 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 $R(\mathcal{T}^r) \leq R(\mathcal{S}^r) + O(c) = R(S) + O(c)$, where $c$ is the number of significant prefixes covering $t^i_j$.*

**Proof.** Let SA be the suffix array of $\mathcal{S}^r$ and SA$'$ the suffix array of $\mathcal{T}^r$. We call a suffix starting at $s^i_j$ *moved* if its original significant prefix is no longer its prefix after the mutation. Hence the relative position of a moved suffix in SA$'$ may differ from its position in SA.

A moved suffix appearing inside a self-repetition of SA or its target interval can break the self-repetition into two pieces in SA$'$. Each moved suffix can also create a new self-repetition and a new target interval by itself. Finally, a moved suffix can leave a hole in its original self-repetition or target interval, splitting it into two parts. Thus at most 6 new runs can appear for each moved suffix. As there are $c$ moved suffixes, up to $6c$ new runs can be created in SA$'$.

The remaining suffixes are first sorted by their significant prefixes. The mutation may affect the ordering of suffixes sharing the same significant prefix, yet does so in a consistent way creating no new self-repetitions. Hence a single mutation can create no more than $O(c)$ new runs. In Appendix A, we show that the upper bound for new runs is at most $2c+3$. ■

The proof immediately generalizes to other types of mutations. Table 1 summarizes some of them. An insertion adds one new suffix corresponding to no position in the base sequence. The new suffix behaves as a moved suffix in the analysis. When a substring is copied, the suffixes starting at range $S^j_{a,b}$ are duplicated. Of these, only the duplicates corresponding to the newly inserted substring can become moved suffixes.

With respect to the number of runs, insertions and deletions are no worse than point mutations, while copying a substring creates no more new runs than two point mutations. When deleting a prefix of some sequence in the collection, no new runs are created, as no remaining significant prefix contains any part of the deleted prefix.

### 2.2 Expected case properties

**Lemma 8 (Karlin et al., 1983).** *Let $S = S_{1,n}$ be a random text. The expected length of the longest repeated substring is $O(\log_\sigma n)$.*

**Lemma 9.** *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 $S^2S^3 \ldots S^r$. The expected value of $R(\mathcal{T}^r)$ is at most $R(S) + O(s \log_\sigma N)$.*

**Proof.** By Lemma 8, significant prefixes of the initial collection are of length $O(\log_\sigma n)$. For a fully random sequence of length $N$, the expected length of significant prefixes would be $O(\log_\sigma N)$. If we apply random mutations to a repeated collection, the collection gradually turns into a fully random sequence. Hence, $O(\log_\sigma N)$ is an upper bound for the expected length of significant prefixes in the final collection.

TABLE 1. GENERALIZING LEMMA 7 FOR OTHER TYPES OF MUTATIONS

| *Type of mutation* | *New $S^i$* | *Suffix moved if* |
|---|---|---|
| *Insertion* of character $c$ after $s^i_a$ | $S^i_{1,a}cS^i_{a+1,n}$ | $s^i_{a+1}$ in significant prefix |
| *Deletion* of $S^i_{q,b}$ | $S^i_{1,a-1}S^i_{b+1,n}$ | $s^i_a$ in significant prefix |
| *Copying* of $S^j_{a,b}$ after $S^i_c$ | $S^i_{1,c}S^j_{a,b}S^i_{c+1,n}$ | $s^i_{c+1}$ or $s^j_{b+1}$ in significant prefix |

The number of moved suffixes limits the number of new runs created by the mutation.

As in the proof of Lemma 7, we call a suffix $T^i_{j,n}$ moved if its significant prefix is not a prefix of $S^i_{j,n}$. This means that at least one mutation has occurred in the significant prefix of $T^i_{j,n}$. As there are $s$ mutations, there are at most $O(s \log_\sigma N)$ moved suffixes in the expected case. The result follows by the same reasoning as in the proof of Lemma 7. ■

The results generalize to other types of mutations as well, as noted in the previous section after Lemma 7.

*2.2.1. Substring-repeated collections.* Instead of a repeated collection, we could use a somewhat more general model.

**Definition 10.** *The r-times* substring-repeated collection *of base text* $S = S_{1,n}$ *is* $\mathcal{S}^r = S^1 S^2 \ldots S^r$, *where* $S^1 = S_{1,n-1}\#$, $S^i = S_{\alpha i,\beta i}\#$ *for all* $i < r$, *and* $S^r = S_{\alpha r,\beta r}\$$, *where* $1 \le \alpha_i \le \beta_i < n$ *for all i.*

Note that any repeated collection of $S$ is also a substring-repeated collection of $S$. The theorems of previous subsections can be generalized for substring-repeated collections in a straightforward manner.

**Theorem 11.** *Let* $r \ge 1$ *and* $\mathcal{S}^r$ *be a substring-repeated collection of* $S = S_{1,n}$ *with total length N. Then,* $R(\mathcal{S}^r) \le R(S) + O(r \log_\sigma N)$.

**Proof.** By Lemma 6, the corresponding repeated collection has $R(S)$ runs. As noted after Lemma 7, the deletion of a prefix creates no new runs and the deletion of a suffix is no worse than a point mutation. Hence the result follows by similar arguments as in the proof of Theorem 9. ■

By similar reasoning, we get the following theorem.

**Theorem 12.** *Let* $\mathcal{S}^r$ *be a substring-repeated collection of a random text* $S = S_{1,n}$ *with total length N. Let* $\mathcal{T}^r$ *be* $\mathcal{S}^r$ *after* $s \ge r$ *mutations at random positions in* $s^2 S^3 \ldots S^r$. *Then the expected value of* $R(\mathcal{T}^r)$ *is at most* $R(S) + O(s \log_\sigma N)$.

In the sequel, we assume w.l.o.g. $s \ge r$, since the case of texts in collection identical to substrings of $T^1$ is non-interesting, and also easy to handle separately within the same space and time bounds that will be achieved.

## 3. BASE STRUCTURES FOR COUNTING QUERIES

Next we will describe three different solutions to support counting queries, whose space will depend on the number of runs in the BWT.

### *3.1. Run-length encoded wavelet tree*

The *wavelet tree* (Grossi et al., 2003) is a binary tree structure that can be used to solve $rank_c(L,i)$ queries for a string $L$ and any $c \in \Sigma$ during backward searching (see Section 1.3). Given a string $L_{1,n}$ from an alphabet of size $\sigma$, its wavelet tree is defined recursively as follows. The root corresponds to the whole sequence $L_{1,n}$. In a *balanced* wavelet tree, the left child (resp. right child) of the root is a wavelet tree of the sequence $L_<$ (resp. $L_\ge$) obtained by concatenating all symbols $l_i < \sigma/2$ (resp. $l_i \ge \sigma/2$). This subdivision is represented by a bit vector $B_{1,n}$ that has value $B[i] = 0$ if symbol $l_i$ belongs to the left subtree, and $B[i] = 1$ otherwise. Recursion stops when the concatenated sequence is a repeat of one symbol. Entropy-bound dictionary structures for bit vectors (Pagh, 2001; Raman et al., 2002) can be used to represent the wavelet tree of $L = S^{bwt}$ in $nH_k(S) + o(n \log \sigma)$ bits of space for any $k \le \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$ (Mäkinen and Navarro, 2007).

Function $rank_c(L,i)$ gives the number of times the symbol $c$ appears in prefix $L_{1,i}$. It can be solved by traversing the wavelet tree starting from the root and calculating *rank* on bit vectors: at each step, choose the left subtree if $c < \sigma/2$, or the right subtree otherwise, and update $i \leftarrow rank_0(B,i)$ or $i \leftarrow rank_1(B,i)$, respectively. When reaching a leaf, the current $i$ value is the answer. In the balanced wavelet tree, this

happens after $O(\log \sigma)$ steps. Since *rank* for binary sequences takes constant time (Pagh, 2001; Raman et al., 2002), $rank_c(L,i)$ is solved in $O(\log \sigma)$ time. In a similar manner, it is possible to recover any symbol $s_i$ from the wavelet tree: We descend left or right depending on $B[i]$ and updating $i$ as before; the leaf reached corresponds to $s_i$.

To make wavelet trees more suitable for repetitive sequence collections, we introduce a data structure that we call *Run-Length encoded Wavelet Tree* (RLWT). Let $R$ be the number of runs in the BWT $L_{1,n}$. Let $B^{all}$ be a level-wise concatenation of all bit vectors in the wavelet tree of $L$. In the worst case, each run in $L$ corresponds to one $0/1$-bit run on each of the $\log \sigma$ levels of the wavelet tree, thus there are at most $R \log \sigma$ $0/1$-bit runs in $B^{all}$. Let $b \leq \lceil \frac{1}{2} R \log \sigma \rceil$ be the number of 1-bit runs in $B^{all}$. The RLWT encodes $B^{all}$ into two separate bit vectors $B^1$ and $B^{rl}$ such that the number of 1-bits in both bit vectors is exactly $b$: bit vector $B^1$ marks all the starting positions of 1-bit runs in $B^{all}$, and bit vector $B^{rl}$ encodes the run-lengths of these runs in *unary coding*. More precisely, $B^1[i] = 1$ only if $B^{all}[i] = 1$ and $B^{all}[i-1] = 0$ for all $1 < i \leq N \log \sigma$, and $B^1[1] = 1$ if $B^{all}[1] = 1$. The unary code for a run of length $j$ contains $j-1$ zero-bits concatenated with one 1-bit. The length of $B^{rl}$ is the sum of the length of 1-bit runs in $B^{all}$, which is at most $N \log \sigma$ bits.

The value of $rank_1(B^{all}, i)$ can be computed from the bit vectors $B^1$ and $B^{rl}$ as follows. Notice that

$$rank_1(B^{all}, i) = rank_1(B^{all}, i - rank_1(B^{all}, j-1) + rank_1(B^{all}, j-1)$$

holds for any $j$. Let $r = rank_1(B^1, i)$ be the run that precedes, or starts at, position $i$, and let $j$ be its starting position, say $j = select_1(B^1, r)$, which denotes the position of the $r$th 1 in $B^1$ and can be solved in constant time within the same space of *rank* (Raman et al., 2002). From the definition of $B^{rl}$ follows that the number of 1-bits preceding position $j$ equals

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

It remains to compute $rank_1(B^{all}, i) - rank_1(B^{all}, j-1)$: Let $k$ be the length of the $r$th run, say $k = select_1(B^{rl}, r) - rank_1(B^{all}, j-1)$. The number of 1-bits in the closed interval $[j, i]$ of the bit vector $B^{all}$ is

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

Finally, the sum of values $rank_1(B^{all}, j-1)$ and $rank_1(B^{all}, i) - rank_1(B^{all}, j-1)$ gives the answer for $rank_1(B^{all}, i)$. Of course $rank_0(B^{all}, i) = i - rank_1(B^{all}, i)$.

With a similar reasoning, $select_1(B^{all}, i) = select_1(B^1, r) + i - select_1(B^{rl}, r)$, where $r = rank_1(B^{rl}, i)$. For $select_0(B^{all}, i)$ we would need analogous structures $B_0^1$ and $B_0^{rl}$, but this is not needed for our indexes.

Thus, we only need to provide binary *rank* and *select* on $B^1$ and $B^{rl}$. The following theorem gives a way to represent them succinctly.

**Theorem 13 (Gupta et al., 2006).** *Given a bit vector $B$ of length $u$ containing $b$ 1-bits, a* binary searchable dictionary representation (BSD) *requires $gap(B) + O(b \log \log(u/b))$ bits of space where $gap(B)$ is at most $b \log(u/b)$ bits. It supports* rank *and* select *queries in $O(\log b)$ time, or* rank *in $t_{\mathrm{BSD}}(b,u)$ time and* select *in $O(\log \log b)$ time by adding $O((b \log(u/b))/ \log b)$ further bits of space, for a total of $b \log(u/b)(1 + o(1)) + O(b \log \log(u/b) + \log u)$. Here*

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

*which is, for example, $O(\sqrt{\log b})$ and $o(\log \log u)^2$.*

For the bit vectors $B^1$ and $B^{rl}$, we have strict upper bounds of $u \leq N \log \sigma$ and $b \leq \left\lceil \frac{1}{2} R \log \sigma \right\rceil$, thus we obtain the following theorem:

**Theorem 14.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the sequences $T^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. The RLWT data structure for the collection takes*

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

*bits of space, where $\sigma$ is the alphabet size. It can compute LF($i$) in time $t_{\mathrm{LF}} = t_{\mathrm{BSD}}(R \log \sigma, N \log \sigma) \cdot \log \sigma$ and supports operation count($P$) in time $O(|P| t_{\mathrm{LF}})$.*

**Proof.** Solving $rank(B^{all}, i)$ requires $t_{\mathrm{BSD}}$ time on the compressed representations of $B^1$ and $B^{rl}$, thus wavelet tree operations $rank_c$ and $t_i^{bwt}$ are supported in $t_{\mathrm{LF}} = O(t_{\mathrm{BSD}} \log \sigma)$ time. Backward searching requires $O(|P|)$ applications of these operations, as well as $\sigma \log N$ additional bits for table $C$ (see Section 1.3). For simplixity, the $O(\log(N \log \sigma))$ terms from the BSD structures are rounded to $O(\sigma \log N)$. ■

The RLWT structure can be made dynamic by using dynamic bit-vector representations for $B^1$ and $B^{all}$. To retain a space usage almost independent of $N$, we use the following representation.

**Theorem 15.** *Given a bit vector $B$ of length $u$ containing $b$ 1-bits, the operations rank, select, insert($B$, $i$, $j$) and delete($B$, $i$) can be supported in $b \log(u/b)(1+o(1)) + O(b \log\log(u/b))$ bits of space and in*

$$t_{\mathrm{DynB}}(b,u) = O\left(\log b + \frac{\log u \cdot \log^{\star} u}{\log b + \log\log u}\right)$$

*time, where insert($B$, $i$, $j$) inserts $j \in \{0,1\}$ between $B[i]$ and $B[i+1]$ and delete($B$, $i$) deletes $B[i]$.*[4]

**Proof.** We build on top of the dynamic gap-encoded structures presented in previous work (Mäkinen and Navarro, 2008). Distances of consecutive 1-bits are $\delta$-encoded (see Section 3.3), and partitioned into superblocks of $\beta = \log u \cdot \log^* u$ bits so that no code is broken. Superblocks are then stored as the leaves of a red-black tree that is augmented to support *rank*/*select* queries (Mäkinen and Navarro, 2008). The tree requires $O((b \log(u/b)/\beta) \log u) = o(b \log(u/b))$ bits of space. To process superblocks in $O(\log u \cdot \log^* u/(\log b + \log\log u))$ time, we use lookup tables that process chunks of $\frac{1}{2}(\log b + \log\log(u/b) - \log\log\log(u/b))$ bits (whose size is sublinear in $b \log(u/b)$). ■

Plugging the dynamic bit vector representation inside the RLWT leads to the following theorem.

**Theorem 16** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1,N]$ of all the sequences $T^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. The dynamic RLWT data structure for the collection takes*

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

*bits of space, where $\sigma$ is the alphabet size. It computes LF($i$) within time $t_{\mathrm{LF}} = t_{\mathrm{DynB}}(R \log \sigma, N \log \sigma) \cdot \log \sigma$, and supports operations count($P$) in time $O(|P| t_{\mathrm{LF}})$, and insert($T$) and delete($T$) in time $O(|T| t_{\mathrm{LF}} + \log r)$.*

**Proof.** Again, since binary *rank* and *select* on $B^1$ and $B^{rl}$ are supported in time $O(t_{\mathrm{DynB}})$, we have binary *rank* on $B^{all}$ within the same time, and hence operations $rank_c$ and $t_i^{bwt}$ in time $t_{\mathrm{LF}} = O(t_{\mathrm{DynB}} \log \sigma)$, from where backward search achieves time $O(|P| t_{\mathrm{DynB}} \log \sigma)$. The results for inserting and deleting texts are inherited from Chan et al. (2007) and Mäkinen and Navarro (2008), who show how to maintain a dynamic text collection by modifying the dynamic representation of $\mathcal{T}^{bwt}$. The dynamic RLWT can be plugged in their construction to obtain the time bounds above. Additional $O(\sigma \log N)$ bits are needed for a binary search tree replacing table $C$. The handle-mechanism (Mäkinen and Navarro, 2008) occupies $O(r \log N)$ bits (another search tree) and stores the mapping of the texts into $T^{bwt}$. This structure is accessed on insertions and deletions, adding (negligible) $O(\log r)$ time. ■

[4]Here $\log^* u$ denotes the iterated logarithm of $u$, that is, the number of times we have to take log to $u$ successively to make it $\leq 1$.

### 3.2. Improved run-length FM-index

Recall that the 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)* (Mäkinen and Navarro, 2005) uses a reduction such that the starts of equal letter runs of $L$ are marked in a bit-vector $E[1, N]$, where $E[i] = 1$ iff $L[i]$ starts a run, and a reduced sequence $L'[1, R]$ is formed, where $L'[rank_1(E, i)] = L[i]$ for $i$ such that $E[i] = 1$. Another bit-vector $D$ is formed that encodes the $LF$-mapping of entries of $L$ marked in $E$: $D[LF(i)] = 1$ iff $E[i] = 1$. It also uses array $C^E[1, \sigma]$, where $C^E[c]$ stores the number of runs of symbols smaller than $c$ in $L$. It can be shown (Mäkinen and Navarro, 2005) that

$$\begin{aligned} LF(i) &= C[L[i]] + rank_c(L, i) \\ &= select_1(D, C^E[L[i]] + rank_{L[i]}(L', rank_1(E, i))) \\ &\quad + (i - select_1(E, rank_1(E, i))). \end{aligned}$$

Value $rank_c(L, i)$ can derived from above (the formula is slightly different when $L[i] \neq c$ (Mäkinen and Navarro, 2005)), and also $L[i] = L'[rank_1(E, i)]$. This enables FM-index functionality.

The original proposal (Mäkinen and Navarro, 2005) uses $2N + o(N)$ bits for $E$ and $D$, $\sigma \log N$ bits for $C^E$, and $R \log \sigma\ (1 + o(1))$ bits for the wavelet tree of $L'$. This can now be improved using the BSD representation for $E$ and $D$ (Theorem 13), giving immediately the following result, called RLFM+.

**Theorem 17.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the sequences $\mathcal{T}^i \in \mathcal{C}$, and $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. The RLFM+ data structure for the collection takes*

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

*bits of space, where $\sigma$ is the alphabet size. It computes $LF(i)$ in $t_{\mathrm{LF}} = \log \sigma \,/\, \log \log R + t_{\mathrm{BSD}}(R, N)$ time, and count$(P)$ in time $O(|P| t_{\mathrm{LF}})$, by using the multiary wavelet tree representation for $R$ (Ferragina et al., 2007).*

We can replace all static structures with dynamic ones to obtain the following.

**Theorem 18.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the sequences $\mathcal{T}^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then the dynamic RLFM+ data structure for the collection takes*

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

*bits of space, where $\sigma$ is the alphabet size. It computes $LF(i)$ within time $t_{\mathrm{LF}} = \log R \log \sigma \,/\, \log \log R + t_{\mathrm{DynB}}(R, N)$, supports operation count$(P)$ in time $O(|P| t_{\mathrm{LF}})$, and operations insert$(T)$ and delete$(T)$ in time $O(|T| t_{\mathrm{LF}} + \log r)$.*

**Proof.** Using the most efficient dynamic wavelet tree representation (González and Navarro, 2008) we have $rank_c$ and $t_i^{bwt}$ (and $select_c$) on $L'$ in time $O(\log R \lceil \log \sigma / \log \log R \rceil)$ and $R \log \sigma (1 + o(1))$ bits of space. We have binary *rank* and *select* on $E$ and $D$ in time $O(t_{\mathrm{DynB}}(R, N))$ using Theorem 15, using $2R \log \frac{N}{R}(1 + o(1)) + O\left(R \log \log \frac{N}{R}\right)$ bits of space. Arrays $C$ and $C^E$ are replaced by binary search trees occupying $O(\sigma \log N)$ bits and giving access in $O(\log \sigma)$ time. This gives $t_{\mathrm{LF}}$ and the counting complexity. The mechanism to maintain a dynamic collection via dynamic $T^{bwt}$ is identical to the one in the proof of Theorem 16, explaining the remaining space/time bounds. ■

### 3.3. Run-length compressed suffix array

The *Run-Length Compressed Suffix Array (RLCSA)* is based on the CSA of Mäkinen et al. (2004). Function $\Psi$ is stored in a compressed form by run-length encoding the differences $\Psi(i) - \Psi(i - 1)$. Fast

access to $\Psi$ is implemented by sampling absolute $\Psi(i)$ values at a rate depending on the desired time-space trade-off.

To encode the run $\Psi(i)\Psi(i+1)\cdots\Psi(i+l)$, we write two integers: the gap since the previous run (or a sampled value) $\Psi(i)-\Psi(i-1)$ and the length of the run, $l+1$. We use delta codes (Elias, 1975) to encode the integers. Let $b(p)$ be the binary representation of $p$ and $b^-(p)$ the same representation without the most significant bit. The encoding of a positive integer $p$ is the binary string $0^{|b(t)|-1}b(t)b^-(p)$, where $t=|b(p)|$. The length of the code is

$$\delta(p) = \log' p + 2\log'\log' p - 2 \tag{2}$$

bits, where $\log' x = |b(x)| = \lceil \log(x+1) \rceil$.

Let $\Psi_c$ be the range of SA corresponding to the suffixes starting with character $c$. Within the range, the values of $\Psi$ form a strictly increasing sequence. To bound the total length of the codes, we note that the sum of differences $\Psi(i)-\Psi(i-1)$ inside each range $\Psi_c$ is at most $N$. Hence the sum of all the $R$ gaps between the runs of $\Psi$ is at most $\sigma N$. Similarly, the total length of the $R$ runs is $N$. Hence the $\Psi$ array requires

$$|\Psi| = R\left(\delta\left(\frac{\sigma N}{R}\right) + \delta\left(\frac{N}{R}\right)\right) \tag{3}$$

bits of space in the worst case to encode the gaps and the run lengths, respectively. This is is justified by the concavity of logarithm, making the worst case to be the one where the gaps are all $\approx \sigma N/R$ and the lengths of runs are all $\approx N/R$.

We sample the first $\Psi(i)$ value of each $B$-bit block of the compressed $\Psi$ array. As we also start a new block whenever the first character of the suffix changes, we have $n_B \leq |\Psi|/B + \sigma$ blocks in the array. As each sample is a pair $(i,\ \Psi(i))$, the samples take a total of $O(n_B \log N)$ bits. We also need the array $C$ used for the *LF*-mapping to determine the ranges $\Psi_c$.

For counting queries, we use the backward search algorithm in Section 1.3 with the following modifications:

$$(5') \qquad sp \leftarrow \min\{i \in \Psi_s,\ \Psi(i) \geq sp\};$$
$$(6') \qquad ep \leftarrow \max\{i \in \Psi_s,\ \Psi(i) \leq ep\};$$

These lines are implemented by binary searching the samples and decoding the correct block. This takes $O(\log n_B)$ time for the binary search and $O(B)$ time for the decoding. As the lines are repeated for each character of the pattern, the query takes $O(|P|(\log n_B + B))$ time.

**Theorem 19.** *Given a collection $\mathcal{C}$ the concatenation $\mathcal{T}[1,N]$ of all the sequences $\mathcal{T}^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then the RLCSA data structure for the collection takes*

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

*bits of space, where $\sigma$ is the alphabet size and $B$ the block size in bits. It supports count($P$) in time $O(|P|t_{\mathrm{LF}})$, where $t_{\mathrm{LF}} = O(\log N + B)$ is the time required to binary search $\Psi$.*

**Proof.** The first term in the size bound come from Equations 2 and 3. Multiplier $1 + O(\log N/B)$ covers the size of the samples. The second term covers array $C$ and the extra samples when the first character of the suffix changes. The $O(\log N)$ term in the time bound is an upper bound for $O(\log n_B)$. ■

We can adapt the techniques used in the dynamic $\Psi$ of Chan and colleagues (Hon et al., 2007; Chan et al., 2007) to construct a dynamic RLCSA. The resulting structure requires $\sigma$ updates per inserted or deleted character, which is clearly worse than the $\log \sigma$ updates in the dynamic RLWT. This is a fundamental difference between the wavelet tree-based indexes and the compressed suffix arrays.

As the $\Psi$ values form a strictly increasing sequence in each range $\Psi_c$, any presentation of those values can be considered a bit vector. This bit vector marks the occurrences of character $c$ in the BWT by 1-bits

(Sirén, 2009). Hence a CSA is essentially a representation of the BWT by $\sigma$ individual bit vectors, whereas a wavelet tree-based index uses only $\log \sigma$ bit vectors to represent the BWT. Because of this difference, a CSA requires $O(1)$ bit vector operations per character for counting and $O(\sigma)$ operations per character for updates, while a wavelet tree-based index requires $O(\log \sigma)$ operations per character for both counting and updates.

## 4. STANDARD SUFFIX ARRAY SAMPLING

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 (Navarro and Mäkinen, 2007) in self-indexes is to sample every $d$-th suffix of each text in the collection in an array $L[1, N/d+1]$, such that $L[i] = SA^{-1}[i \cdot d]$, mark the locations $L[i]$ into a bit-vector $B[1, N]$, such that $B[L[i]] = 1$ for all $1 \le i \le N/d+1$, and store the samples in the suffix array order in a table $S[1, N/d+1]$, such that $S[rank_1(B, L[i])] = i \cdot d$.

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

Function locate($P$) works in a similar fashion; first backward search is applied to find the range SA[$sp$, $ep$] containing the occurrences of the pattern $P$ and SA[$i$] is computed for each $sp \le i \le ep$ as follows. If suffix SA[$i$] is not sampled ($B[i] = 0$), then the $LF$-mapping is applied until an index $j$ is found where SA[$j$] is sampled ($B[j] = 1$). Then $\mathrm{SA}[i] = S[rank_1(B, j)] + c$, where $c < d$ is the number of times $LF$-mapping was applied. This takes time $t_{\mathrm{SA}} = O(t_{\mathrm{LF}} d)$. To finalize locate, we can use a table $K$ such that $k = K[rank_1(B, j)]$ gives the text $T^k$ containing the sampled suffix. Then $j' = S[rank_1(B, j)] - Pos[k] + 1$ is the starting position of the sampled suffix inside $T^k$ and $i' = j' + c$ is the starting position of the suffix SA[$i$] inside $T^k$, unless we have spanned some border of two texts during the $c$ steps of $LF$-mapping. In case we have, since the borders are marked with special characters #, we can nevertheless easily compute the correct text $T^{k'}$ and index $i'$ inside it. Table $Pos$ can be replaced by the BSD structure of Theorem 13 as follows: $Pos[k] = select(D, k)$, where $D$ is a bit-vector with a bit set at the starting position of each text in the concatenation. Bit-vector $D$ can also be used to directly give mapping from SA[$i$] to the relative text position, but the above strategy gives better time/space tradeoff.

These operations are slightly different in compressed suffix arrays. Instead of using $LF$-mapping to move backward in the sequence, function $\Psi$ is used to move forward. This changes display and locate in the obvious way.

The space required by the standard solution is $O(r \log N + (N/d) \log N + N)$ bits, which can be reduced to $O\left(r \log \frac{N}{r} + (N/d) \log N + (N/d) \log \frac{N}{r}\right) = O(r \log n + (N/d) \log N)$ by using Theorem 13; this changes the time for locate into $t_{\mathrm{SA}} = O((t_{\mathrm{LF}} + t_{\mathrm{BSD}}(N/d, N))d)$ plus $O(\log \log r)$. The following theorem summarizes a simplified result relevant to our context, where table $Pos$ is stored as is.

**Theorem 20** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the $r$ sequences $\mathcal{T}^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then there is a data structure for the* repetitive substring collection problem *taking the space of any of the static solutions of Section 3 (i.e., Theorems 14, 17, or 19), plus $O(r \log N + (N/d) \log N)$ bits of space, where $d$ is a parameter. It supports count($P$) in the time of the corresponding theorem of Section 3, locate($P$) in the time for count($P$) plus $O(t_{\mathrm{SA}})$ per occurrence, and display($k$, $i$, $j$) in time $O((d+j-i)t_{\mathrm{LF}})$, where $t_{\mathrm{LF}}$ is the time to compute an LF-step in the solution for counting, and $t_{\mathrm{SA}} = d(t_{\mathrm{LF}} + t_{\mathrm{BSD}}(N/d, N))$. The structure also supports* SA[$i$] *in $O(t_{\mathrm{SA}})$ time,* $\mathrm{SA}^{-1}[(k, j)]$ *(which is a form of* $\mathrm{SA}^{-1}$ *where we give text number $k$ and relative position $j$) within the time to display a symbol, and $\mathcal{T}[\mathrm{SA}[i]]$ in time $O(t_{\mathrm{BSD}}(\sigma, N))$.*

**Proof.** To compute SA[$i$] and to locate, we carry out the described process of computing up to $d$ $LF$-steps and checking whether $B[i] = 1$ in the BSD representation. To finalize locate, we map SA[$i$] to pair ($k$, $j$) as described earlier using tables $Pos$ and $K$. To compute $\mathrm{SA}^{-1}[(k, j)]$ and to display, we follow the next sampled text position to suffix array index and compute up to $d$ $LF$-steps (this time it is known how many steps to compute). For $\mathcal{T}[\mathrm{SA}[i]]$ we could binary search $C$ for SA[$i$]. A faster solution within $O(\sigma \log N)$

bits of space is to represent $C$ as a BSD-compressed bitmap $G[1, N]$ where $G[C[c]] = 1$ for all $c \in \Sigma$, otherwise zero. ■

The result can be made dynamic by using the data structure of Theorem 18 and *dynamic sampling* as described in previous work (Chan et al., 2007; Mäkinen and Navarro, 2008): Instead of tables $S$ and $L$ the samples are stored in binary search trees and the bit-vector $B$ and table $C$ are replaced by dynamic bit-vectors as in Theorem 15. No representation of array $Pos$ is required since it is subsumed by the structure supporting *insert* and *delete*. We obtain the following result.

**Theorem 21.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the $r$ sequences $\mathcal{T}^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then there is a data structure for the* dynamic repetitive substring collection problem *taking the same space of the static structure in Theorem 20. Now the space for counting and the achieved time $t_{\mathrm{LF}}$ correspond to a dynamic structure of Section 3, in particular to Theorems 16 or 18. The structure computes locate($P$) in the time of count($P$) plus $O(t_{\mathrm{SA}} + \log r)$ per occurrence, display($k$, $i$, $j$) in time $O(\log r + (d + j - i)t_{\mathrm{LF}})$, and insert($T$) and delete($T$) in time $O(|T|(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N)) + \log r)$, where $t_{\mathrm{SA}} = d(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N))$. The structure also computes* $\mathrm{SA}[i]$ *in time $O(t_{\mathrm{SA}})$, $\mathrm{SA}^{-1}[(k, j)]$ within the time to display one character, and $\mathcal{T}[\mathrm{SA}[i]]$ in time $O(t_{\mathrm{DynB}}(\sigma, N))$.*

## 5. ADVANCED SUFFIX ARRAY SAMPLING

Our objective is to have locate and display times within $O(polylog(N))$. With the standard sampling approach of the previous section, this holds only if we assume $r = O(polylog(N))$; then $d$ can be chosen as $r \log N$ to achieve space $O((N/d) \log N) = O(n)$, i.e., independent of $N$ as we wish.

We will now show that by exploiting the repetitiveness of the collection, it is possible to achieve better space with time requirements less dependent on $r$.

### 5.1. Improving space and time for display

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

To be able to use the same samples for other texts in the collection, we *align* the other texts $T^2, T^3, \ldots, T^r$ to substrings of $T^1$ and encode the alignment space-efficiently. Let us redefine $Pos[k]$ as the occurrence position of $T^k$ inside $T^1$ with $s_k$ differences, where $\sum_{k=2}^{r} s_k = s$, i.e., $Pos[k] = \alpha_k$ of Def. 3. Let $T^k_{del}$ be the string $T^k$ where we have inserted special symbols $\emptyset$ at the positions where a character of $T^1$ was deleted in order to convert $T^1$ into $T^k$. If there are $d_k \leq s_k$ deletions in this transformation, then $|T^k_{del}| = |T^k| + d_k$. Let $D^k[1, |T^k_{del}|]$ be a bit-vector where the positions of the corresponding deletions operations are marked, $E^k[1, |T^k|]$ a bit-vector where the positions of the inserted and mutated positions of $T^k$ are marked, and $M^k[1, s_k - d_k]$ a bit-vector where the mutations are distinguished from the insertions. Thus $D^k[i] = 1$ iff $T^k_{del}[i] = \emptyset$; $E^k[i] = 1$ iff $T^k[i]$ is obtained from an insertion or a mutation from $T^1$; and if it is, then $M^k[rank_1(E^k, i)] = 1$ iff the edit operation is a mutation. The inserted (mutated) symbols are stored in another array $IM^k[1, s_k - d_k]$ in their order of occurrence in $T^k$.

Consider now a query display($k$, $i$, $j$). The position $j'$ of $T^1$ aligned to $t^k_j$ can be computed by

$$j' = Pos[k] + select_0(D^k, j) - rank_0(M^k, rank_1(E^k, j)) \tag{4}$$

where the number of deletions minus insertions is computed up to position $j$ in $T^k$. The same is done for value $i$, and next 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 edit operations stored for $T^k$ can also be extracted using *rank* on the corresponding bit-vectors. More precisely, we set a finger at the end of the virtual string $T^k_{del}[select_0(D^k, i), select_0(D^k, j)]$. Each time we advance backwards in $T^1_{i',j'}$, we consider the current position $p$ in $T^k_{del}$. If $D^k[p] = 1$, we ignore the character extracted from $T^1$, and shift $p$ and compute $LF$ on $T^1$. Else, let $p' = rank_0(D^k, p)$. If $E^k[p'] = 0$, we output the extracted symbol from $T^1$, and shift $p$ and compute $LF$. Else, let $p'' = rank_1(E^k, p')$, we output $IM[p'']$ and shift $p$. If $M^k[p''] = 1$, then we also compute $LF$.

Note the amount of work is proportional to $j-i+1$, plus the number of deletions made to obtain $T^k[i,j]$ from $T^1$. A way to avoid this is to extract each symbol of $T^k[i,j]$ individually, which requires locating each relevant character in $T^1$. We have obtained the following partial result.

**Theorem 22.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1,N]$ of all the sequences $T^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then there is a data structure for the* repetitive substring collection problem *taking the space of any solution of Section 3 for counting, plus*

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

*bits of space, where $\sigma$ is the alphabet size, $n=|T^1|$, $s$ is the number of edit differences to align $T^2, T^3, \ldots, T^r$ within $T^1$, and $d$ is a parameter. Besides counting, the structure supports display$(k, i, j)$ in time $O(\min((j-i+1)\ d, (d+j-i+s^k_{i,j}))\, t_{\mathrm{LF}})$, where $t_{\mathrm{LF}}$ is that of the counting structure and $s^k_{i,j}$ is the number of deletions to align $T^k_{i,j}$ within $T^1$.*

**Proof.** The largest of the new structures are the bit-vectors $D^k$. All together, they contain at most $s$ 1-bits out of $N-n+s$, so they can each be represented using BSD, leading to the space reported (we assumed the same space for the $E^k$ vectors, which are only slightly smaller). Vector $M^k$ requires only $O(s)$ bits. We also store the edited symbols in $IM$, adding $s\log\sigma$ bits. The other terms correspond to the usual sampling of $T^1$. The time requirement corresponds to either extracting each symbol of $T^1$ individually or to traversing the whole area of $T^1$ consecutively. ■

### 5.2. Improving space and time 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 SA. Hence, once every $d$-th suffix of $T^1$ is sampled, we can reveal any SA[$i$] by applying $LF$-mapping at most $d$ times until finding an entry $j$ such that SA[$j'$] is sampled for some $j'<j$ and $j-j'\le r$. Checking for this is identical to $j-select_1(B,rank_1(B,j))\le r$, where $B$ is the bit-vector marking the locations of the sampled suffixes of $T^1$ in SA. Then 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 the $LF$-mapping was applied, and $k=j-select_1(B,\ rank_1(B,\ j))+1$.

Generalizing the scheme to work under different length texts and with edit differences 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 Theorem 12; class B contains the suffixes whose significant prefixes cover the edit operations and ends of the texts, respectively. Class A contains all other suffixes.

Let us first consider the case when SA[$i$] is a class B suffix. Class B suffixes form at most $s+r$ disjoint regions in texts $T^k$, $2\le k\le r$. 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 $B'[1,\ N]$, table $S'[1,\ rank_1(B',\ N)]$ stores these sampled suffixes in the order they appear in SA, and table $K'[1,\ rank_1(B',\ N)]$ stores the sequence numbers where the sampled suffixes belong to. Retrieving SA[$i$] is completely analogous to the standard sampling scheme by using $S'$, $K'$, and $B'$ in place of $S$, $K$, and $B$, and some representation for array $Pos$ (for mapping SA[$i$] to the relative position). We simply apply $LF$ until $B'[i]=1$ and then find the answer at $S'[rank_1(B',\ i)]$ (as we cannot know whether we are in a class-B or class-A zone, this test is done in parallel to that for solving class-A suffixes, see next). The average space for the samples is bounded by $O(((s\ \log_\sigma N)/d)\ \log N)$ by Theorem 12.

Computing SA[$i$] for class-A suffixes is more challenging than in the case of $r$ identical texts, since now some sampled suffixes of $T^1$ will not have counterparts in all the other texts. We would need somehow access to a list $Q[rank_1(B, \mathrm{SA}^{-1}[i\cdot d])] = k_1k_2\cdots k_p$ denoting texts $T^{k_1}, T^{k_2}, \ldots, T^{k_p}, p\le r$, which have a suffix aligned to $T^1_{i\cdot d,n}$ occurring consecutively around $T^1_{i\cdot d,n}$ in the suffix array. The exact positions inside the texts can be easily restored using the structures storing the alignment for display queries (see Section 5.1).

Before facing the space issue with the lists, let us first consider how to use them if available. In addition to the lists, we need to have for each sampled suffix $T^1_{i \cdot d,n}$ its lexicographic rank $e$ among the suffixes in the list $Q[rank_1(B, \mathrm{SA}^{-1}[i \cdot d])] = k_1 k_2 \cdots k_p$, that is, $e$ such that $k_e = 1$. Now, consider again the situation where SA[$i$] belongs to class A and $LF$ mapping has brought us to entry SA[$j$]. Let us compute $prev = select_1(B, rank_1(B, j))$, $succ = select_1(B, rank_1(B, j) + 1)$, $dprev = j - prev$, and $dsucc = succ - j$. 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 SA[$j$] belongs. This follows directly from the definition of lists $Q[\cdot]$. Analogously, one can check whether SA[$j$] belongs to the list $Q[rank_1(B, succ)] = k_1 k_2 \cdots k_{p'}$ of SA[$succ$]: if $dsucc < e'$, then $k_{e' - dsucc}$, where $e'$ is such that $k_{e'} = 1$. These are the two cases that can happen, and after at most $d$ steps of $LF$-mapping the correct $Q$-list is found.

Now we are left with a space problem: 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 by modifying a classical solution by Overmars (1981) to *$k^{th}$ element/rank searching in the past*. Let us first review the original solution (with slight changes to suite our purposes) and then show how to make the solution more space-efficient and *confluental persistent* (for background on persistent data structures, see Kaplan, 2005).

**Definition 23.** *Let $E(t) = e^t_1 e^t_2 \cdots e^t_{p_t} \in \mathcal{R}^* = \{1, 2, \ldots, r\}^*$ 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 $E(tprev)$, $tprev = max\{t' \in H,\ t' < t\}$, by* deleting *some $e^{tprev}_k$ or* inserting *a new element $e \in \mathcal{R}$ between some $e^{tprev}_{k-1}$ and $e^{tprev}_k$ (or before $e^{tprev}_1$ or after $e^{tprev}_{p_t}$). Let $E(0)$ be an empty sequence. The* persistent selection *problem is to construct a static data structure $\mathcal{D}$ on $\{E(t), t \in H\}$ that supports operation access$(t, k) = e^t_k$. 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($H$)*; if $t = \max(H)$, then tprev is taken as t and sequence $E(t)$ is modified accordingly without retaining its previous version. The* confluental persistent selection *problem allows value t to be any $t \in \mathcal{H}$ also for insertions and deletions; if also $t \in H$, then tprev is taken as t and sequence $E(t)$ is modified accordingly without retaining its previous version.*

The online persistent selection problem allows online updates, but the past remains static, whereas in the confluental selection problem the insertions and deletions can be understood as changes to the past; the effect cumulates to all time points that take place after the change.

**Theorem 24 (Overmars, 1981).** *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 access(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 insert and delete operations executed during the lifetime of $\mathcal{D}$.*

**Proof.** 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 supports 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 $e^t_k$. It is easy to find that leaf by following the path from the root and comparing $k$ with the sum of subtree sizes of left children of traversed nodes. 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 of Overmars (1981) is to copy the old subtree size information into a new field on each node on the path and increment that field. 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 the 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 (Overmars, 1981). 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 25.** *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 access(t, k) in $O(\log x)$ time, and insert(t, e, k) and delete(t, k) in amortized $O(\log x)$ time. There is also an online/confluental version of $\mathcal{D}$ that occupies the same space, but access(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 24 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 a succinct data structure for (dynamic) partial sums to support operations $sum(\hat{S}^v, i) = \sum_{j=1}^{i} \hat{s}_j^v = s_i^v$. 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 (that is, the pointers to children associated to each new field), as we have the connection $sum(\hat{S}^v, i) = sum(\hat{S}^l, i - i') + sum(\hat{S}^r, i')$, where $i' = rank_1(B^v, i)$, and $l$ and $r$ are the left and right children of $v$. That is, $sum(\hat{S}^l, i - i')$ and $sum(\hat{S}^r, i')$ are the subtree sizes of nodes $l$ and $r$, respectively, at the same time point of $s_i^v$. 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 *sum* on $\hat{S}^v$ and dynamic *rank* on $B^v$.

Let us consider how to provide $sum(\hat{S}^v, i) = s_i^v$. First notice 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 $G^v = 10^{f(\hat{s}_1^v)-1} 10^{f(\hat{s}_2^v)-1} \cdots 10^{f(\hat{s}_{k^v}^v)-1}$. Then $sum(\hat{S}^v, i)$ equals $2 \cdot rank_1(F^v, j-1) - (j-1)$, where $j = select_1(G^v, i+1)$. That is, $\sum_{v \in \mathcal{T}} (|F^v| + |G^v|)(1 + o(1)) = O(x \log x)$ bits is enough to support constant time *access* on all subtree sizes, when the tree is static. In the dynamic case, *access* takes $O(\log x)$ time (Blanford and Blelloch, 2004). 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 their operation costs $O(1)$ or $O(\log x)$ per node depending on the case. The associated values in the leaves occupy $O(x \log r)$ bits. ■

Combining Lemma 9 and the RLFM+ structure of Section 3.2 with Theorem 25 applied to sampling gives us the main result of the paper:

**Theorem 26.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the r sequences $T^i \in \mathcal{C}$, there is a data structure for the* repetitive substring collection indexing problem *taking*

$$\begin{aligned}
&\left(R \log \sigma + 2R \log \frac{N}{R}\right)(1 + o(1)) + O\left(R \log \log \frac{N}{R}\right) \\
&+ 3s \log \frac{N - n + s}{s}(1 + o(1)) + O\left(s \log \log \frac{N - n + s}{s}\right) \\
&+ s \log \sigma + O((n/d) \log N) + O(r \log n) + O(\sigma \log N) \\
&+ O(((s \log_\sigma N)/d) \log N) + O(s \log s)
\end{aligned}$$

*bits of space in the average case, where $\sigma$ is the alphabet size, $n = |T^1|$, $s \geq r$ is the number of edit differences to align aligning $T^2, T^3, \ldots, T^r$ within $T^1$, and $d$ is a parameter. The structure computes LF within time $t_{LF} = \log \sigma / \log \log R + t_{BSD}(R, N)$ and supports count(P) in time $O(|P| t_{LF})$, locate(P) after count(P) in time $O(d(t_{LF} + t_{BSD}(N/d, N)) + \log s)$ per occurrence, and display(k, i, j) in time $O(\min((j - i + 1)d, (d + j - i + s_{i,j}^k)) t_{LF})$. It also computes* $\mathrm{SA}[i]$ *and* $\mathrm{SA}^{-1}[(k, d)]$ *in time $t_{SA} = O(d(t_{LF} + t_{BSD}(N/d, N)) + \log s)$, and $\mathcal{T}[\mathrm{SA}[i]]$ in time $O(t_{BSD}(\sigma, N))$.*

**Proof.** The first line of the space corresponds to the counting structure of Theorem 17, and the next two lines, essentially, to the display structures of Theorem 22. These provide the stated time complexities for *LF*, count and display. The time for $\mathcal{T}[\mathrm{SA}[i]]$ is shown in Theorem 20.

According to the discussion preceding persistent selection, we have a bitmap $B$ with $n/d$ bits set out of $N$ for suffixes of type A and a bitmap $B'$ with $(s \log_\sigma \mathrm{N})/d$ expected bits set out of $N$ for suffixes of type B. Both BSD representations are considered in the space requirements and in the time for locate, as we must

check both $B$ and $B'$ after each $LF$ step. After at most $d$ steps, we have either fell close to a sampled suffix of $T^1$, or we have definitely entered a B-zone, all of which are $d$-sampled (we have chosen a worst-case upper bound for both BSD times). For sampled B-zone suffixes, we use BSD representation $D$ for the array $Pos$, to obtain better space (its slower access time will be subsumed by other terms).

What is left is how to complete the locate operation for type-A suffixes once we fall close to a sampled suffix of $T^1$. This is what is provided with the $Q$-lists $Q[1], Q[2], \ldots, Q[n/d]$, which Theorem 25 represents using persistent select: We can regard $T^1$ as a right-to-left timeline. Assume its final position is sampled, then $E(n)$ is just a list with the single element 1. As we read backwards, the $r-1$ texts aligned to $T^1$ appear and disappear from the lists, generating $2(r-1) = O(s)$ insertions/deletions. Each of the $s$ edit operations makes the text disappear from the list, as its suffixes become of type B; and the text reappears when the significant prefix does not cover the mutation anymore. These are the other $2s$ insertions/deletions that arise (Fig. 2). Thus the theorem applies with $h = n/d$, $x = O(s)$, $r = r \leq s$, hence the space is $O(s(\log s + \log(n/d))$ and the access time to an element of any list is $O(\log s)$. This is the time to select the $k$-th text aligned to a sampled suffix, which is done as the final step when locating a suffix of type A.

Computation of $\mathrm{SA}[i]$ is identical to locate, but computation of $\mathrm{SA}^{-1}[(k, j)]$ needs some interplay with the structures of Theorem 25, considered next.

In the case $t_j^k$ belongs to an area where a sampled position is at distance $d$, computation of $\mathrm{SA}^{-1}[(k, j)]$ resembles the display operation. Thus the first attempt is to try up to $d$ $LF$-steps looking for a sampled position in $B$ or $B'$. This covers the cases where $t_j^k$ is within a B-zone or in $T^1$, which have their own explicit sampling. Otherwise we are within an A-zone, and therefore can map $j$ to the corresponding position $j'$ in $T^1$ using the structures of Section 5.1. Now we find $\mathrm{SA}^{-1}[(1, j')]$ in at most $d$ $LF$-steps, as $T^1$ is regularly sampled. The desired answer $\mathrm{SA}^{-1}[(k, j)]$ is now very close, at most $r$ positions away. The final problem is how to find it nearby $\mathrm{SA}^{-1}[(1, j')]$. Since we explicitly store, for the $Q$ list under consideration, the rank $e$ of text $T^1$ in the list, our problem is equivalent to finding the rank of text $T^k$ in the persistent tree of Theorem 25.

Whenever a new leaf is added to the persistent tree, we 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 in a bitmap aligned to $T^2 \ldots T^r$, using $s \log \frac{N-n}{s}(1 + o(1)) + O(s \log \log \frac{N-n}{s} + \log N)$ bits, so that one can find the closest location to $(k, j)$ having a pointer, using $rank$ in $t_{\mathrm{BSD}}(s, N)$ 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. This rank is valid at the latest edit position $j^-$ in $T^k$ preceding $j$. Several edits at other texts may occur between these two positions in the persistent structure, so we wish to update the rank of $T^k$ in the persistent

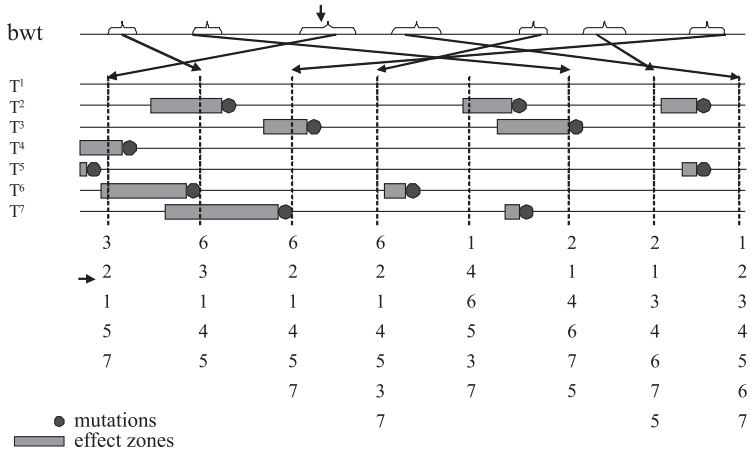


**FIG. 2.** Persistent selection and changes in the lexicographic order of sampled suffixes. Text $T^1$ has been sampled regularly and the pointers to the sampled suffixes are stored with respect to the BWT sequence. Each such pointer is associated a range containing the occurrences of the same significant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

structure between $j^-$ and $j$. This rank is indeed the sum of $O(\log r)$ subtree sizes. The evolution of the subtree size of each such node $v$ can be tracked by using $select_1$ on $G^v$ and $rank_1$ on $F^v$, the bit-vectors of Theorem 25. Thus the time to find this position is $O(\log s + t_{\mathrm{BSD}}(s, N) + \log r) = O(\log s)$. The overall time is the same as for computing SA[$i$]. ■

We recall that on average $R = n \min(1, H_k(T^1)) + \sigma^k + O(s \log_\sigma N)$ for any $k$, thus the following simplified upper bound holds for the space of our structure:

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

By replacing static with dynamic structures we obtain the following result.

**Theorem 27.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1, N]$ of all the sequences $T^i \in \mathcal{C}$, there is a data structure for the* dynamic repetitive substring collection problem *taking the same space of the static structure in Theorem 26. The structure computes LF in time $t_{\mathrm{LF}} = \log \mathrm{N}\lceil \log \sigma \,/\, \log \log N \rceil$, and supports count($P$) in time $O(|P| t_{\mathrm{LF}})$, locate($P$) after count($P$) in time $O(d(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N)) + \log {}^2 s)$ per occurrence, display($k$, $i$, $j$) in time $O(\min((j - i + 1)d, (d + j - i + s^k_{i,j}))\, t_{\mathrm{LF}})$, insert($T$) (given the alignment to $T^1$) in average time $O(|T|(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N)) + s^T \log \sigma \log {}^2 s)$, and delete($T$) in time $O(|T|(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N)) + s^T \log^2 s)$, where $s^T$ is the number of edit operations to align $T$ with some substring of $T^1$. The structure also supports* SA[$i$] *and* $\mathrm{SA}^{-1}[(k, j)]$ *in time $t_{\mathrm{SA}} = O(d(t_{\mathrm{LF}} + t_{\mathrm{DynB}}(N/d, N)) + \log^2 s)$, and $\mathcal{T}$*[SA[$i$]] *in time $O(t_{\mathrm{DynB}}(\sigma, N))$.*

**Proof.** For the absolute samples, the construction is analogous to Theorem 21. The alignment is easy to store in dynamic data structures to support display. For locate we need the confluental persistent select of Theorem 25. The insertions to the structure take average time $O(s^T \log_\sigma N \log^2 s)$, because one has to access the structure on each suffix of class B to determine the class; once the first suffix of class A is found preceding a mutation, the checking can be omitted until the next mutation. The deletions to the structure take $O(s^T \log^2 s)$ time as the boundaries between class A and class B suffixes are known. ■

## 6. FULLY-COMPRESSED SUFFIX TREES

In this section, we introduce a fully-compressed suffix tree for repetitive sequence collections. Let us start with the basic definitions and with the original solution.

The *suffix tree* $\mathcal{S}$ of a text $T_{1,n}$ is a compact trie storing all the suffixes $T_{i,n}$ where the leaves point to the corresponding $i$ values (Apostolico, 1985; Gusfield, 1997). For technical convenience, we assume that $T$ is terminated with a special symbol, so that all lexicographical comparisons are well defined. For a node $v$ in $\mathcal{S}$, $\pi(v)$ denotes the string obtained by reading the edge-labels when walking from the root to $v$, or the *path-label* of $v$ (Russo et al., 2008a). The *string-depth* of $v$ is the length of $\pi(v)$. We will simulate suffix tree behaviour by an implementation of the following abstract data structure, which supports more functionality than the concrete suffix tree.

**Definition 28.** *A* suffix tree representation *supports the following operations:*

- ROOT()*: the root of the suffix tree.*
- LOCATE($v$)*: the suffix position $i$ if $v$ is the leaf of suffix $T_{i,n}$, otherwise* NULL.
- ANCESTOR($v$, $w$)*: true if $v$ is an ancestor of $w$.*
- SDEPTH($v$)/TDEPTH($v$)*: the string-depth/tree-depth of $v$.*
- COUNT($v$)*: the number of leaves in the subtree rooted at $v$.*
- PARENT($v$)*: the parent node of $v$.*
- FCHILD($v$)/NSIBLING($v$)*: the alphabetically first child/next sibling of $v$.*
- SLINK($v$)*: the* suffix-link *of $v$; i.e., the node $w$ such that $\pi(w) = \beta$ if $\pi(v) = a\beta$ for a $a \in \Sigma$.*
- $\mathrm{SLINK}^i(v)$*: the* iterated suffix-link *of $v$; (node $w$ such that $\pi(w) = \beta$ if $\pi(v) = \alpha\beta$ for $\alpha \in \Sigma^i$).*
- LCA($v$,$w$)*: the* lowest common ancestor *of $v$ and $w$.*

- CHILD($v,a$)*: the node $w$ such that the first letter on edge $(v,w)$ is $a \in \Sigma$.*
- LETTER($v$, $i$)*: the $i$th letter of $v$'s path-label, $\pi(v)[i]$.*
- LAQS($v$, $d$)/LAQT($v$, $d$)*: the highest ancestor of $v$ with string-depth/tree-depth $\geq d$.*

We call a suffix tree dynamic when can be maintained on a dynamic collection $\mathcal{C}$.

### 6.1. Original solution

Russo et al. (2008a) described a suffix tree representation that needs only $o(n \log \sigma)$ bits of space on top of a compressed suffix array and supports operations in Def. 28 in polylogarithmic time. The solution is based on a *sampled suffix tree*.

**Theorem 29 (Russo et al., 2008a).** *Using a compressed suffix array CSA that supports the functions $\psi$, $\psi^i$, $T$[SA[$v$]] and LF in times $O(t_\psi)$, $O(t_\Phi)$, $O(1)$ and $O(t_{\mathrm{LF}})$, respectively, a suffix tree for the string $T_{1,n}$ can be represented in $|CSA| + O((n/\delta) \log n) + o(n \log \sigma)$ bits of space and with the time complexities given in Table 2.*

The sampled suffix tree $S$ chooses $O(n/\delta)$ nodes from $\mathcal{S}$, ensuring that for each node $v \in \mathcal{S}$ there is an $i < \delta$ such that the node SLINK$^i(v) \in S$ (Russo et al., 2008a). The sampled tree $S$ can be constructed space-efficiently (see Section 6.3). Sampled nodes store pointers and information about SDEPTH and TDEPTH, and the tree is augmented to support $\mathrm{LCA}_S$ in constant time.

Mapping from the suffix tree $\mathcal{S}$ to the $\delta$-sampled tree $S$ is done via two operations: The operation LSA($v$) for a node $v \in \mathcal{S}$ maps into the *lowest sampled ancestor* of $v$ in $S$. The operation LCSA($v$, $v'$) for nodes $v, v' \in \mathcal{S}$ maps into the *lowest common sampled ancestor* of $v$ and $v'$ in $S$. For any leaf node $v \in \mathcal{S}$, LSA ($v$)can be calculated in constant time using a bit vector $B_{1,n}$ that contains $O(n/\delta)$ 1-bits, and an array of $O(n/\delta)$ values (Russo et al., 2008a). Thus, both operations LSA and LCSA can be solved in $O((n/\delta) \log n) + o(n)$ bits of space and in constant time for leaf nodes in $\mathcal{S}$.

Mapping internal nodes, as well as solving the other operations, is done by carrying out $O(\delta)$ steps involving LCSA computation on leaves plus accessing some data explicitly stored at sampled nodes.

Operations TDEPTH, LAQT and LAQS, require another set of $O(n/\delta)$ sampled nodes (Russo et al., 2008a): for any $v \in \mathcal{S}$, there must be a $j < \delta$ such that PARENT$^j(v) \in \mathcal{S}$. It is not known how to generate this sampling space-efficiently; they currently require $O(n \log n)$ construction space. If LAQS is not supported, operations FCHILD and NSIBLING take the same time as operation CHILD.

Computing CHILD requires that sampled nodes store a list of child-nodes and first letters of their edges. To avoid storing $O(\sigma n/\delta)$ integers, they *mark* one out of $\delta$ leaf nodes and consider only the children whose subtree contains marked leaves (Russo et al., 2008a). The amortized space is then $O((n/\delta) \log n)$ bits, and CHILD can be computed in $O(\log \sigma + t_\Phi \log \delta + (t_\psi + t_{\mathrm{LF}})\delta)$ time.

TABLE 2. TIME COMPLEXITIES FOR THE ORIGINAL FCST

| | *FCST* | *Ours* |
|---|---|---|
| SDEPTH/LOCATE | $t_\psi \delta$ | $(t_\psi + t_{\mathrm{BSD}})\delta$ |
| COUNT/ANCESTOR | 1 | 1 |
| PARENT | $(t_\psi + t_{\mathrm{LF}})\delta$ | $(t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| FCHILD/NSIBLING | $(t_\psi + t_{\mathrm{LF}})\delta$ | $(t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| SLINK | $(t_\psi + t_{\mathrm{LF}})\delta$ | $(t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| SLINK$^i$ | $t_\Phi + (t_\psi + t_{\mathrm{LF}})\delta$ | $t_\Phi + (t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| LETTER | $t_\Phi$ | $t_\Phi$ |
| LCA | $(t_\psi + t_{\mathrm{LF}})\delta$ | $(t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| CHILD | $\log \sigma + t_\Phi \log \delta + (t_\psi + t_{\mathrm{LF}})\delta$ | $\log \sigma + t_\Phi \log \delta + (t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |
| TDEPTH | $(t_\psi + t_{\mathrm{LF}})\delta^2$ | $(t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta^2$ |
| LAQT | $\log n + (t_\psi + t_{\mathrm{LF}})\delta^2$ | $\log N + (t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta^2$ |
| LAQS | $\log n + (t_\psi + t_{\mathrm{LF}})\delta$ | $\log N + (t_\psi + t_{\mathrm{BSD}} + t_{\mathrm{LF}})\delta$ |

Time complexities for the original FCST (Russo et al., 2008a) for a string $T_{1,n}$, and our version for a repetitive collection of length $N$, where $t_{\mathrm{BSD}} = t_{\mathrm{BSD}}(N/\delta, N)$.

The overall space of the fully-compressed suffix tree is $|CSA| + O((n/\delta)\log n) + o(n) = |CSA| + o(n \log \sigma)$ bits if $\delta = \omega(\log_\sigma n)$. Using the FM-index of Ferragina et al. (2007) as the compressed suffix array, the space required is $nH_k + o(n \log \sigma)$ bits. Table 2 summarizes the time complexities of the suffix tree operations.

### 6.2. Indexing repetitive sequence collections

We build on the data structure just described. The space requirement of the sampled tree is $O((N/\delta)\log N)$ bits, which can be made arbitrary small by growing $\delta$ (thus trading time for space). To support LSA and LCSA we represent bit vector $B$ (see Section 6.1) using Theorem 13 in $O((N/\delta)\log \delta)$ bits, thus avoiding the large $o(N)$ term. Now computing LSA and LCSA takes $t_{\mathrm{BSD}}(N/\delta, N)$ time. All the rest is inherited verbatim.

**Theorem 30.** *Given a collection $\mathcal{C}$ and a compressed suffix array that requires $|CSA|$ bits of space and supports the functions $\psi$, $\psi^i$, $T[\mathrm{SA}[v]]$ and LF in times $O(t_\psi)$, $O(t_\Phi)$, $O(t_{\mathrm{BSD}}(\sigma, N))$ and $O(t_{\mathrm{LF}})$, respectively, a suffix tree $\mathcal{S}$ for a concatenated sequence $\mathcal{T}$ of all the $r$ sequences $T^i \in \mathcal{C}$ can be represented in $|CSA| + O((N/\delta)\log N)$ bits of space with the time complexities given in Table 2 (right). Here we assume $t_{\mathrm{BSD}}(\sigma, N) = O(t_\Phi)$.*

By combining the above with Theorems 20 and 17, we immediately obtain the following.

**Corollary 31.** *Given a collection $\mathcal{C}$ and the concatenation $\mathcal{T}[1,N]$ of all the $r$ sequences $T^i \in \mathcal{C}$, let $R$ be the number of runs in the BW-transformed sequence $\mathcal{T}^{bwt}$ of $\mathcal{T}$. Then the suffix tree for $\mathcal{T}$ can be represented in*

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

*bits of space, where $\sigma$ is the alphabet size and $\delta$ is a parameter. The structure achieves the time complexities in Table 2 (right), with $t_{\mathrm{LF}} = \frac{\log\sigma}{\log\log R} + t_{\mathrm{BSD}}(R,N)$, $t_\psi = \frac{\log\sigma}{\log\log R} + t_{\mathrm{BSD}}(R,N) + t_{\mathrm{BSD}}(\sigma,N)$, $t_\phi = \delta\left(\frac{\log\sigma}{\log\log R} + t_{\mathrm{BSD}}(R,N) + t_{\mathrm{BSD}}(N/\delta,N)\right)$, and $\mathcal{T}[\mathrm{SA}[v]]$ in $O(t_{\mathrm{BSD}}(\sigma, N))$ time.*

**Proof.** Theorems 20 and 17, with $d = \delta$, provide directly $t_{\mathrm{LF}}$ and $\mathcal{T}[\mathrm{SA}[v]]$. Also, $t_\phi = \delta\left(\frac{\log\sigma}{\log\log R} + t_{\mathrm{BSD}}(R,N) + t_{\mathrm{BSD}}(N/\delta,N)\right)$ as $\psi^i[j] = \mathrm{SA}^{-1}[\mathrm{SA}[j]+i]$. Finally, for $\Psi$ we use the formula (Lee and Park, 2007) $\Psi(i) = select_{\mathcal{T}[\mathrm{SA}[i]]}(\mathcal{T}^{bwt}, i - C[\mathcal{T}[\mathrm{SA}[i]]])$. Operation *select* is supported by our wavelet tree in time $O\left(\frac{\log\sigma}{\log\log R}\right)$ (Ferragina et al., 2007), and the RLFM scheme (Mäkinen and Navarro, 2005) can be easily adapted to compute *select* on $L = \mathcal{T}^{bwt}$ via *select* on the (standard) wavelet tree of $L'$ plus a constant number of accesses to bit-vectors $D$ and $E$, recall Section 3.2.

A simplified formula is, for example, $t_\psi = t_{\mathrm{LF}} = O(\log R + \log\sigma)$, $t_\Phi = O(\delta \log N)$, and $\mathcal{T}[\mathrm{SA}[i]]$ in time $O(\log\sigma)$. Notice that if $r = polylog(N)$, one can choose $\delta = r \log N \log\log N$ to make term $O((N/\delta)\log N)$ disappear, and still all suffix tree operations are supported in $polylog(N)$ time. ■

### 6.3. Dynamic fully-compressed suffix tree

A dynamic suffix tree representation that needs $o(n \log\sigma)$ bits of space on top of a dynamic CSA was described by Russo et al. (2008b). It supports operations in Def. 28 excluding TDepth, LAQt and LAQs, in polylogarithmic time. The solution is based on a dynamic version of the $\delta$-sampled suffix tree described in Section 6.1. The operations stay the same.

**Theorem 32 (Russo et al., 2008b).** *Given a dynamic CSA that supports functions $\psi$, $\psi^i$, $\mathcal{T}[\mathrm{SA}[v]]$ and LF in times $O(t_\psi)$, $O(t_\Phi)$, $O(\log N)$ and $O(t_{\mathrm{LF}})$, respectively, a suffix tree for a dynamic collection of length $N$ can be represented in $|CSA| + o(N \log\sigma)$ bits of space and with the time complexities given in Table 3.*

**Updating the sampled tree.** The $\delta$-sampled tree $S$ of the suffix tree $\mathcal{S}$ is represented with a dynamic parentheses structure by Chan et al. (2007). This structure supports $\mathrm{LCA}_S$ and $\mathrm{Parent}_S$ in $O(\log N)$ time

TABLE 3. TIME COMPLEXITIES FOR THE DYNAMIC FCST

| | *Dynamic FCST and ours* |
|---|---|
| SDEPTH/LOCATE | $t_\psi\delta$ |
| COUNT/ANCESTOR | 1 |
| PARENT | $(t_\psi + t_{\mathrm{LF}})\delta$ |
| SLINK | $(t_\psi + t_{\mathrm{LF}})\delta$ |
| SLINK$^i$ | $t_\Phi + (t_\psi + t_{\mathrm{LF}})\delta$ |
| LETTER | $t_\Phi$ |
| LCA | $(t_\psi + t_{\mathrm{LF}})\delta$ |
| CHILD/FCHILD/NSIBLING | $(t_\psi + t_{\mathrm{LF}})\delta + t_\Phi \log \delta + (\log N)\log(N/\delta)$ |
| INSERT($T$)/ DELETE($T$) | $\lvert T\rvert(t_\psi + t_{\mathrm{LF}})\delta$ |

Time complexities for the dynamic FCST (Russo et al., 2008b) and our version for a dynamic collection $\mathcal{C}$ of length $N$.

and can be augmented to support SDEPTH in $O(\log N)$ time. The $\delta$-sampled tree is maintained while inserting the text to the CSA: at every step for any $v \in \mathcal{S}$, there is an $i < \delta$ such that SLINK$^i(v)$ is sampled (Russo et al., 2008b). Hence, space-efficient construction of the $\delta$-sampled tree is done by inserting a text into a empty collection. These structures require $O((N/\delta) \log N)$ bits of space, and the time to INSERT/DELETE nodes to/from $S$ is dominated by the time to compute SLINK.

Operations LSA and LCSA are supported in $O(\log N)$ time and $O((N/\delta) \log \delta) + o(N)$ bits of space. This is achieved by using a dynamic bit vector to represent $B[1, N]$, which contains $O(N/\delta)$ 1-bits (Russo et al., 2008b). The operations TDEPTH, LAQT and LAQS are not supported in the dynamic FCST. The operation CHILD is computed by *generalized branching* in time $O((t_\psi + t_{\mathrm{LF}})\delta + t_\Phi \log \delta + (\log N) \log(N/\delta))$ (Russo et al., 2008b). Because LAQS is not supported, operations FCHILD and NSIBLING take the same time as the operation CHILD.

**Indexing repetitive collections.** We only need to replace their dynamic bit vector representation by that of Theorem 15, so that it takes $O((N/\delta) \log N)$ bits of space while supporting operations LSA and LCSA in $t_{\mathrm{DynB}}(N/\delta, N) = O(\log N)$ time. The time complexities of the original solution do not worsen.

**Corollary 33.** *Given a collection $\mathcal{C}$, a dynamic suffix tree for a concatenated sequence $\mathcal{T}$ of all the $r$ sequences $T^i \in \mathcal{C}$ can be represented within the same space of Corollary 31. The structure achieves the time complexities in Table 3, with* $t_{\mathrm{LF}} = \frac{\log \sigma}{\log \log R}$ , $\log R + t_{\mathrm{DynB}}(R, N), t_\psi = \frac{\log \sigma}{\log \log R} \log R + t_{\mathrm{DynB}}(R, N) + t_{\mathrm{DynB}}(\sigma, N), t_\phi = \log r + \delta\left(\frac{\log \sigma}{\log \log R} \log R + t_{\mathrm{DynB}}(R, N) + t_{\mathrm{DynB}}(N/\delta, N)\right)$, *and $\mathcal{T}$[SA[$v$]] in $O(t_{\mathrm{DynB}}(\sigma,N))$ time.*

**Proof.** Theorems 21 and 18 provide the running times analogously as in the proof of the static version. ■

Simplified formulas are obtained using $t_{\mathrm{DynB}}(\cdot, N) = O(\log N)$.

## 7. IMPLEMENTATION AND EXPERIMENTS

We have implemented the static base structures considered in this paper for solving the repetitive substring collection problem.[5] Some preliminary versions of the dynamic structures exist as well, but here we limit the experiments to the static structures. The compressed suffix trees considered in this paper use almost verbatim the mechanisms developed in the original papers, so once those implementations are available, it will be easy to plug in our base structures.

We first experiment with the static base structures for counting (Section 3) coupled with the standard sampling techniques (Section 4). These combinations can be used to add reasonably efficient support for

[5]See www.cs.helsinki.fi/group/suds/rlcsa for the source code of the RLCSA structure that turns out to offer the best time/space tradeoffs.

TABLE 4. BASE STRUCTURE SIZES AND TIMES FOR COUNT AND DISPLAY

| *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 | 41.34 | 1.24 | 0.70 |
| RLWT | 35.06 | 4.49 | 3.01 |
| RLFM+ | 53.08 | 2.10 | 1.55 |

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 (www.sanger.ac.uk/Teams/Team71/durbin/sgrp/). $\Psi$ sampling rate was set to 128 in CSA and to 32 bytes in RLCSA. Reported times are in microseconds/character.

the repetitive substring collection problem, as shown in Table 4 and Figure 3. The Compressed Suffix Array (CSA) (Sadakane, 2003), Succinct Suffix Array (SSA) (Mäkinen and Navarro, 2005; Ferragina et al., 2007), and Run-Length FM-index (RLFM) (Mäkinen and Navarro, 2005) 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 4 GB of RAM (3.2 GB visible to OS) running Fedora Core 8 based Linux. Counting and locating times are averages over 1000 patterns of length 10. Displaying times were calculated by extracting the whole text from the indexes.

The new structures for display and locate (Section 5) require the alignment of each sequence with the base sequence to be given. Our implementation supports alignments with insertions, deletions, and substitutions as described in the theory part. In addition, runs of don't cares (i.e., Ns) are treated separately so that the display structure does not grow too big with these artificial 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 25, as we discard completely the tree structure and instead concatenate levelwise the two bit vectors stored at the nodes of the tree; a third bit vector 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 $6x \log x(1 + o(1)) + x \log \frac{h}{x}(1 + o(1)) + O\left(x \log \log \frac{h}{x}\right) + x \log r$ bits, where $6x \log x(1 + o(1))$ comes from the 3 bit vectors of length $x$ supporting *rank* and *select* on each of the at most 2 log $x$ levels of

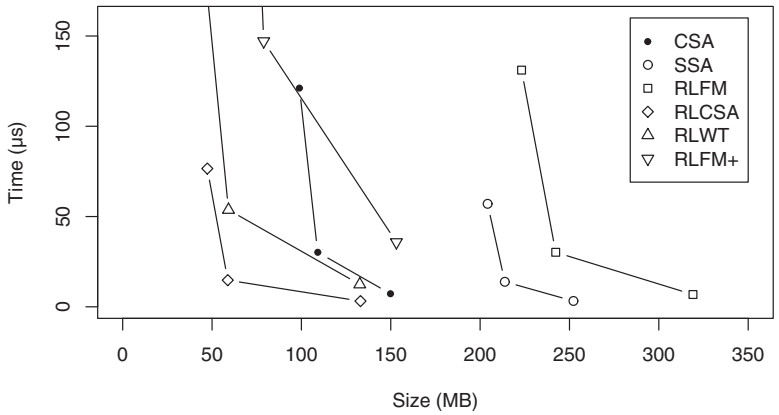


**FIG. 3.** Sizes and times for locate for self-indexes on the S paradoxus collection. Each index was tested with sampling rates $d = 32$, 128, and 512. Reported times are in microseconds/occurrence.

TABLE 5. STANDARD SAMPLING VERSUS PERSISTENT SELECTION

| | *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 |

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.

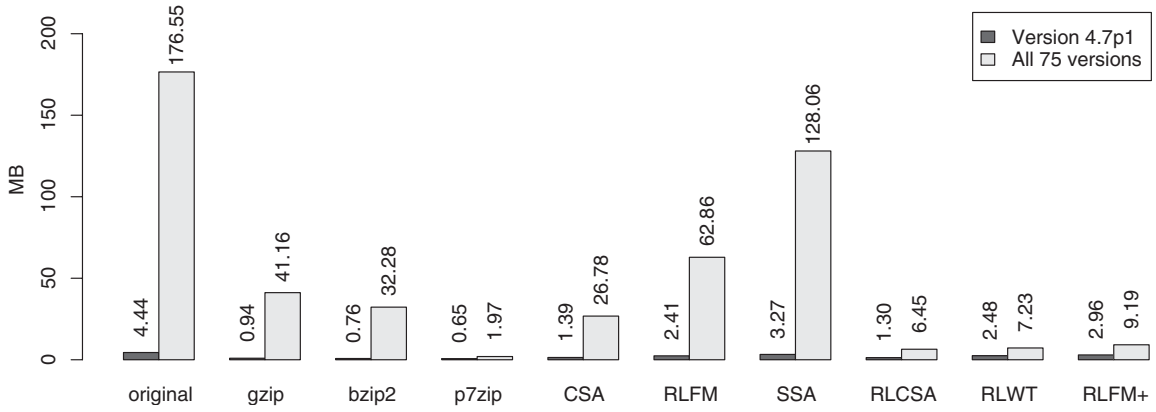


**FIG. 4.** Compression results for OpenSSH sources.

the red-black balanced tree, $x \log \frac{h}{x}(1 + o(1)) + O\left(x \log \log \frac{h}{x}\right)$ 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 Figure 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 occupied 3.97 MB.[6] 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 amounts of random mutations on it. Table 5 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 (0.0001). This made the running times reasonably close; for example, one locate took 172 versus 184 microseconds on 0.0001 mutation rate, respectively.

In order to test our indexes in other applications where repetitive collections arise, we focus on version control systems. We test our indexes on the source code for portable versions of OpenSSH,[7] namely, on a 4.44 MB tar archive containing the source code for version 4.7p1, as well as on another 176.55 MB archive containing the source code for all 75 versions up to version 4.7p1. The latter contains multiple copies of the

[6]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.

[7]www.openssh.com.

same files as well as many highly similar files, making it highly compressible. We compare our indexes with existing self-indexes as well as plain compressors. In addition to the state-of-the-art compressor p7zip,[8] we use the well-known gzip[9] and bzip2[10] compressors with parameter $-9$. Due to their small block sizes, gzip and bzip2 give an idea of the traditional entropy-based compressibility of the collection. As seen in Figure 4, our indexes clearly outperform the existing self-indexes. Again RLWT outperforms RLFM+ even with this larger alphabet size, indicating that the average RLWT space requirement is better than the worst case.

## 8. 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 both theoretical and practical variants that have shown to be much more successful.

We have focused on the base structures that support only limited functionality themselves, but on top of which one can build more flexible structures. In particular, we have demonstrated that compressed suffix trees can exploit the new base structures to achieve improved space for highly repetitive sequence collections.

Our base structures can also be used directly to support more advanced search functionalities such as regular expression search and approximate search, using the general backward backtracking framework (Lam et al., 2008; Langmead et al., 2009; Li and Durbin, 2009) as discussed in Section 1.3.

The experiments showed that the theoretically best solution for the display and locate operations (Section 5) is inferior to the solution that uses standard sampling (Section 4), on mutation levels expected across individual genomes. Still, our development on persistence has independent interest and could find applications in other areas where lower mutation rates are common. The good part in the superiority of the standard sampling approach is its universality; the data structures do not need to know what and where the mutations are, nor to identify any base sequence; the mutations just become part of the analysis. The structures can hence be used on any sequence collection without knowing their alignment. The analysis captures most of the typical mutation types such as insertions, deletions, substitutions, and translocations. Reversals can be captured by the analysis by adding the reverse complement of the chosen base sequence to the collection.

It is hence a plausible scenario to have a massive highly repetitive collection of sequences, such as 1000 Human Genomes, represented as one self-index residing in main memory. Such tool could support very fast extraction of any part of any sequence and very fast exact search with any given pattern. In addition, the tool would support approximate searching to the extent required e.g. in the current short read mapping applications.

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 (p7zip). 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.

[8] http://p7zip.sourceforge.net.
[9] www.gzip.org.
[10] www.bzip.org.

## 9. APPENDIX

### A. Number of runs created per mutation

We are going to improve the $R(\mathcal{T}^r) \leq R(S) + 6c$ bound for the number of runs in Lemma 7 to $R(S) + 2c + 3$. To do this, we assume there are $c$ moved suffixes, starting at positions $i$ to $j$ in the concatenated sequence $\mathcal{T}$. We denote these suffixes as $\mathcal{T}_i$ to $\mathcal{T}_j$. We will present a run as two ranges of suffix array values,

$$(\ldots, a, b, c, \ldots) \rightarrow (\ldots, a+1, b+1, c+1, \ldots),$$

where the left side is the self-repetition and the right side is its target interval.

As noted in the proof, there are six ways a moved suffix can create new runs:

1. Split a self-repetition into two pieces by moving inside it.
2. Create a new self-repetition by itself.
3. Split a target interval into two pieces by moving inside it.
4. Create a new target interval by itself.
5. Leave a hole in the self-repetition in its original position.
6. Leave a hole in the target interval in its original position.

Each moved suffix can create at most one new run for each of the cases.

Consider the following run in the suffix array before the mutation:

$$(\ldots, a, i-1, b, \ldots) \rightarrow (\ldots, a+1, i, b+1, \ldots)$$

As $\mathcal{T}_i$ moves while $\mathcal{T}_{i-1}$ does not, we end up with three runs: $(\ldots, a) \rightarrow (\ldots, a+1)$ as the original run, $(i-1) \rightarrow (i)$ from case 4 for $\mathcal{T}_i$, and $(b, \ldots) \rightarrow (b+1, \ldots)$ from case 6 for $\mathcal{T}_i$.

Case 6 does not happen for other moved suffixes. If $\mathcal{T}_{k+1}$ leaves a hole in a target interval, $\mathcal{T}_k$ leaves a corresponding hole in the self-repetition. Hence, a run like

$$(\ldots, c, k, d, \ldots) \rightarrow (\ldots, c+1, k+1, d+1, \ldots)$$

becomes

$$(\ldots, c, d, \ldots) \rightarrow (\ldots, c+1, d+1, \ldots).$$

Symmetrically, case 5 can happen only for $\mathcal{T}_j$, and if it happens, we also get case 2 for $\mathcal{T}_j$. Hence we get at most 4 new runs from the cases covered so far.

Consider next cases 2 and 4 for the rest of the moved suffixes. Assume $\mathcal{T}_k$ moves and does not become a part of either of the self-repetitions surrounding its new position. This makes $\mathcal{T}_k$ a new self-repetition of length 1. But that means $\mathcal{T}_{k+1}$ becomes the target interval of the newly created run. Hence the new run created by case 2 for $\mathcal{T}_k$ is the same run as the one created by case 4 for $\mathcal{T}_{k+1}$. As we are left with case 2 for suffixes $\mathcal{T}_i$ to $\mathcal{T}_{j-1}$, and case 4 for suffixes $\mathcal{T}_{i+1}$ to $\mathcal{T}_j$, we can get at most $c-1$ new runs from them. This makes the total so far at most $c+3$.

Finally, we have cases 1 and 3. Recall that we are dealing with runs in BWT, which translates into restricted runs in $\Psi$, where a self-repetition can only contain suffixes starting with the same character.

Assume we have the run $(\ldots, a, b, \ldots) \rightarrow (\ldots, a+1, b+1, \ldots)$. Suffix $\mathcal{T}_k$ moves inside the run, and the self-repetition becomes $(\ldots, a, k, b, \ldots)$. As we are dealing with restricted runs, suffixes $\mathcal{T}_a$ and $\mathcal{T}_b$ must start with the same character. This means that $\mathcal{T}_k$ must also start with the same character. But in that case, we must have $\mathcal{T}_{a+1} < \mathcal{T}_{k+1} < \mathcal{T}_{b+1}$, so the target interval becomes $(\ldots, a+1, k+1, b+1, \ldots)$, and case 1 does not happen for $\mathcal{T}_k$.

We get at most $c$ new runs from case 3. As case 1 cannot happen, and we get at most $c+3$ new runs from the other cases, a point mutation can create at most $2c+3$ new runs.

## 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 to the yeast genome collection. V.M. was funded by the Academy of Finland (grant 119815). G.N. was partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB; grant ICM P05-001-F, Mideplan, Chile). J.S. was funded by the Research Foundation of the University of Helsinki and Academy of Finland (grant 119815). N.V. was funded by the Helsinki Graduate School in Computer Science and Engineering. Preliminary/partial versions of this article have appeared in SPIRE 2008 and in RECOMB 2009.

## DISCLOSURE STATEMENT

No competing financial interests exist.

## REFERENCES

Apostolico, A. 1985. The myriad virtues of subword trees, 85–96. *Combinatorial Algorithms on Words. NATO ISI Series.* Springer-Verlag, New York.

Blanford, D., and Blelloch, G. 2004. Compact representations of ordered sets. *Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms* 11–19.

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

Chan, H.-L., Hon, W.-K., Lam, T.-W., et al. 2007. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms* 3, 2.

Church, G.M. 2006. Genomes for all. *Sci. Am.* 294, 47–54.

Cover, T., and Thomas, J. 1991. *Elements of Information Theory*. Wiley, New York.

Elias, P. 1975. Universāl codeword sets and representations of the integers. *IEEE Trans. Inform. Theory* 21, 194–203.

Ferragina, P., and Manzini, G. 2005. Indexing compressed texts. *J. ACM* 52, 552–581.

Ferragina, P., Manzini, G., Mäkinen, V., et al. 2007. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* 3, 2.

Fischer, J., Mäkinen, V., and Navarro, G. 2008. An(other) entropy-bounded compressed suffix tree. *Lect. Notes Comput. Sci.* 5029, 152–165.

Giancarlo, R., Scaturro, D., and Utro, F. 2009. Textual data compression in computational biology: a synopsis. *Bioinformatics* (advance access).

González, R., and Navarro, G. 2008. Improved dynamic rank-select entropy-bound structures. *Lect. Notes Comput. Sci.* 4957, 374–386.

Grossi, R., and Vitter, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.* 35, 378–407.

Grossi, R., Gupta, A., and Vitter, J. 2003. High-order entropy-compressed text indexes. *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms* 841–850.

Grümbach, S., and Tahi, F. 1993. Compression of DNA sequences. *Proc. 13th Data Compression Conf.* 340–350.

Grümbach, S., and Tahi, F. 1994. A new challenge for compression algorithms: genetic sequences. *Inform. Process. Manage.* 30, 875–886.

Gupta, A. Hon, W.-K., Shah, R., et al. 2006. Compressed data structures: dictionaries and data-aware measures. *Proc. 16th Data Compression Conf.* 213–222.

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

Hall, N. 2007. Advanced sequencing technologies and their wider impact in microbiology. *J. Exp. Biol.* 209, 1518–1525.

Hon, W.-K., Lam, T.-W., Sadakane, K., et al. 2007. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica* 48, 23–36.

Kaplan, H. 2005. *Handbook of Data Structures and Applications*. Chapman & Hall, New York.

Karlin, S., Ghandour, G., Ost, F., et al. 1983. New approaches for computer analysis of nucleic acid sequences. *Proc. Natl. Acad. Sci. USA* 80, 5660–5664.

Lam, T. W., Sung, W. K., Tam, S. L., et al. 2008. Compressed indexing and local alignment of DNA. *Bioinformatics* 24, 791–797.

Langmead, B., Trapnell, C., Pop, M., et al. 2009. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol.* 10, R25.

Lee, S., and Park, K. 2007. Dynamic rank-select structures with applications to run-length encoded texts. *Proc. 19th Annu. Symp. Combin. Pattern Match.* 95–106.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 707–710.

Li, H., and Durbin, R. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 1754–1760.

Mäkinen, V., and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic J. Comput.* 12, 40–66.

Mäkinen, V., and Navarro, G. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms* 4, 32.

Mäkinen, V., Navarro, G., and Sadakane, K. 2004. Advantages of backward searching—efficient secondary memory and distributed implementation of compressed suffix arrays. *Lect. Notes Comput. Sci.* 3341, 681–692.

Mäkinen, V., and Navarro, G. 2007. Implicit compression boosting with applications to self-indexing. *Lect. Notes Comput. Sci.* 4726, 214–226.

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

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *J. ACM* 48, 407–430.

Navarro, G., and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Comput. Surv.* 39, 2.

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

Pagh, R. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.* 31, 353–363.

Pennisi, E. 2007. Breakthrough of the year: human genetic variation. *Science* 21, 1842–1843.

Raman, R., Raman, V., and Rao, S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms* 233–242.

Russo, L., Navarro, G., and Oliveira, A. 2008a. Fully-compressed suffix trees. *Lect. Notes Comput. Sci.* 4957, 362–373.

Russo, L., Navarro, G., and Oliveira, A. 2008b. Dynamic fully-compressed suffix trees. *Lect. Notes Comput. Sci.* 5029, 191–203.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms* 48, 294–313.

Sadakane, K. 2007. Compressed suffix trees with full functionality. *Theory Comput. Syst.* 41, 589–607.

Sirén, J. 2009. Compressed suffix arrays for massive data. *Proc. 16th SPIRE*. Springer LNLS 5721, 63–74.

Address correspondence to:
*Dr. Veli Mäkinen*
*Department of Computer Science*
*University of Helsinki*
*P.O. Box 68*
*FI-00014 Helsinki*
*Finland*

*E-mail:* vmakinen@cs.helsinki.fi