Compressed Suffix Trees for Repetitive Texts ⋆

Andrés Abeliuk[1,2] and Gonzalo Navarro[1]

[1]Department of Computer Science, University of Chile.
[2]Instituto de Filosofía y Ciencias de la Complejidad, IFICC.
{aabeliuk,gnavarro}@dcc.uchile.cl

Abstract. We design a new compressed suffix tree specifically tailored to highly repetitive text collections. This is particularly useful for sequence analysis on large collections of genomes of the close species. We build on an existing compressed suffix tree that applies statistical compression, and modify it so that it works on the grammar-compressed version of the longest common prefix array, whose differential version inherits much of the repetitiveness of the text.

1 Introduction

The suffix tree [27, 20] is arguably the most beautiful and relevant data structure for string analysis. It has been said to have a myriad virtues [1], and it has a myriad applications in many areas, particularly bioinformatics [11]. A suffix tree built on a text T supports pattern matching in T in time proportional to the length of the pattern. In addition, many complex sequence analysis problems are solved through sophisticated traversals over the suffix tree. Thus, a suffix tree implementation must support a variety of navigation operations. These involve not only the classical tree navigation (parent, child) but also specific ones such as suffix links and lowest common ancestors.

One of the main drawbacks of suffix trees is their considerable space requirement, which is usually close to $20n$ bytes for a text of n symbols, and at the very least $10n$ bytes [14]. For example, the human genome, containing approximately 3 billion bases, could easily fit in the main memory of a desktop computer (as each DNA symbol needs just 2 bits). However, its suffix tree would require 30 to 60 gigabytes, too large to fit in normal main memories. A way to reduce this space to about 4 bytes per symbol is to use a simplified structure called a suffix array [18], which still offers pattern matching but misses important suffix tree operations such as suffix links and lowest common ancestor operations. This reduces the relevance of suffix arrays in many biological problems, whereas in many other areas suffix arrays are sufficient.

Much research on compressed representations of suffix trees and arrays, which operate in compressed form, has been pursued. Progress has been made in terms of the *statistical* compressibility of the text collection, that is, how biased are the symbol frequencies given a short context of k symbols around them.

⋆ Partially funded by Fondecyt grant 1-080019 and Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

A recent challenge raised by the sharply falling costs of sequencing[1] is the growth of large sequence databases formed by genomes of individuals of the same or closely related species. In a few years, biologists will need to handle databases of thousands to millions of genomes: sequencing machines are already producing the equivalent of thousands of genomes per day[2]. These requirements dwarf the current challenges of indexing *one* genome in main memory, and could never be tackled with statistical compression based techniques.

Fortunately, these huge databases have a feature that renders them tractable: they are *highly repetitive*. Two human genomes share 99.9% of their sequences, for example. Such features are not captured by statistical compression methods (i.e., the frequencies of symbols stay roughly the same in a database of many near-copies of the same sequence). Rather, we need *repetition aware* compression methods. Although this kind of compression is well-known (e.g., grammar-based and Ziv-Lempel-based compression), only recently there have appeared compressed suffix arrays and other indexes capable of pattern searching that take advantage of repetitiveness [17, 5, 4, 13]. Yet, none of the existing compressed suffix trees [26, 8, 7, 23, 25, 9], is tailored to repetitive text collections.

Our contribution is to present the *first* compressed suffix tree with full functionality, whose compression effectiveness is related to the repetitiveness of the text collection. While our operations are much slower than existing compressed suffix trees, the space required is also much lower on repetitive collections.

2 Our Contribution in Context

Most of the research in this area is focused on compressed suffix arrays [22] (CSAs, generically), whose functionality is not sufficient for many computational biology problems. There are, however, various recent results showing how to enhance a CSA in order to obtain a compressed suffix tree (CST, generically) [26, 8, 7, 23, 25, 9]. Essentially, they show that if one adds longest common prefix (LCP) information, one can obtain a CST from a CSA.

The first CST was Sadakane's [26]. Apart from the CSA, it used $2n$ bits to represent the LCP, plus other $4n$ bits to support navigation. Most operations are supported in constant time. The best existing implementation [9] shows that it uses about 13 bpc (bits per character) and very few microseconds per query.

The second proposal was by Russo et al. [25]. It requires only $o(n)$ bits on top of a CSA, and the operations are supported in polylogarithmic time. The implementation achieved very little space, around 4.5 bpc. However, operations take order of milliseconds.

A third proposal by Fischer et al. [8], later improved by Fischer [7], achieves $(1/\epsilon)nH_k$ extra bits, where H_k is the k-th order empirical entropy of T (a measure of statistical compressibility [19]), for any constant $\epsilon > 0$. Operation times

[1] See `http://www.guardian.co.uk/world/feedarticle/10038353`

[2] See `http://www.nytimes.com/2011/12/01/business/dna-sequencing-caught-in-deluge-of-data.html?_r=2`

are of the form $O(\log^\epsilon n)$. Different practical variants of this structure were designed and implemented by Cánovas and Navarro [3] and by Ohlebusch, Gog and Fischer [23, 9]. The best implementations use as little as 10 bpc and execute operations in a few microseconds (but usually slower than Sadakane's CST).

We introduce a new CST that, for current repetitive biological collections, reaches 1.3–1.5 bpc, in exchange for operation times in the order of milliseconds. This large space difference with previous CSTs should widen on larger repetitive collections (i.e., thousands or more similar sequences, as opposed to tens in our test collections), whereas current CSTs would simply grow linearly in size.

Our result is built on three blocks, which will be detailed in the next sections:

1. We build on the only existing CSA that is tailored to repetitive collections, the Run-Length CSA (RLCSA) [17]. The size of the RLCSA is a function of the number of *runs in* Ψ, a concept that will be explained in Section 4 and that is related to the repetitiveness of the text collection. The RLCSA will be responsible for most of the final space, 0.85–0.95 bpc.
2. We use a base $2n$-bit LCP representation that was initially proposed by Sadakane [26]. Fischer et al. [8] showed that this sequence could be represented using a number of bits that is, again, related to the runs in Ψ (see Section 3. Cánovas and Navarro [3] showed empirically that the compression achieved was insignificant on non-repetitive collections. In this paper we show that, on repetitive collections, this idea does pay off, adding just 0.2–0.25 bpc to the space.
3. Fischer et al. [8] show how to map all the CST operations into three queries over the LCP array: *range minimum queries* (RMQ) and a new primitive called *next/previous smaller value* (NSV/PSV), see Section 3. We design a novel index on the LCP to answer those queries, inspired on that of Cánovas and Navarro [3], but whose size depends on the number of runs in Ψ. Inspired in a local suffix array compression method [10], we grammar-compress the differential LCP array and replace the regular tree structure used by Cánovas and Navarro by a (pruned) grammar tree resulting from the LCP compression. This index adds about 0.2–0.3 further bpc to the space.

3 Our Base Compressed Suffix Tree

A *suffix array* [18] over a text $T[1, n]$ is an array $A[1, n]$ of the positions in T, lexicographically sorted by the suffix starting at the corresponding position of T. That is, $T[A[i], n] < T[A[i+1], n]$ for all $1 \leq i < n$. Note that every substring of T is the prefix of a suffix, and that all suffixes starting with a given pattern P appear consecutively in A, hence a couple of binary searches find the area $A[sp, ep]$ containing all the positions where P occurs in T.

There are several compressed suffix arrays (CSAs) [22, 6], which offer essentially the following functionality: (1) Given a pattern $P[1, m]$, find the interval $A[sp, ep]$ of the suffixes starting with P; (2) obtain $A[i]$ given i; (3) obtain $A^{-1}[j]$ given j. An important function the CSAs implement is $\Psi(i) =$

$A^{-1}[(A[i] \bmod n)+1]$ and its inverse, usually much faster than computing A and A^{-1} . This function lets us move virtually in the text, from the suffix i that points to text position $j = A[i]$, to the one pointing to $j+1 = A[\Psi(i)]$.

A *suffix tree* [27, 20, 1] is a compact trie (or digital tree) storing all the suffixes of T. This is a labeled tree where each text suffix is read in a root-to-leaf path, and the children of a node are labeled by different characters. Leaves are formed when the prefix of the corresponding suffix is already unique. Here "compact" means that unary paths are converted into a single edge, labeled by the string formed by concatenating the involved character labels. If the children of each node are ordered lexicographically by their string label, then the leaves of the suffix tree form the suffix array of T.

In order to get a suffix tree from a suffix array, one needs the *longest common prefix* (LCP) information, that is, $LCP[i]$ is the length of the longest common prefix between suffixes $T[A[i-1], n]$ and $T[A[i], n]$, for $i > 1$, and $LCP[1] = 0$ (or, seen another way, the length of the string labeling the path from the root to the lowest common ancestor node of suffix tree leaves i and $i-1$). The suffix tree topology is implicit if we identify each suffix tree node with the suffix array interval containing the leaves that descend from it. This range uniquely identifies the node because there are no unary nodes in a suffix tree. A *compressed suffix tree* (CST) is obtained by enriching a CSA with some representation of the LCP data, plus some extra space to support fast queries.

Sadakane [26] showed how to compress the LCP array to just $2n$ bits by noticing that, if sorted by text order rather than suffix array order, the LCP numbers decrease by at most 1. Let $PLCP$ be the permuted LCP array, then $PLCP[j+1] \geq PLCP[j]-1$. Thus the numbers can be differentially encoded, $h[j+1] = PLCP[j+1] - PLCP[j] + 1 \geq 0$, and then represented in unary over a bitmap $H[1, 2n] = 0^{h[1]}10^{h[2]} \ldots 10^{h[n]}1$. Then, to obtain $LCP[i]$, we look for $PLCP[A[i]]$, and this is extracted from H via *rank/select* operations. Here $rank_b(H, i)$ counts the number of bits b in $H[1, i]$ and $select_b(H, i)$ is the position of the i-th b in H. Both can be answered in constant time using $o(n)$ extra bits of space [21]. Then $PLCP[j] = select_1(H, j) - 2j$, assuming $PLCP[0] = 0$.

Fischer et al. [8] prove that array H is compressible, as it has at most $2r$ runs of 0s or 1s. Here, r is the number of *runs in* Ψ, which is related to the repetitiveness of T and will be discussed in Section 4 (the more repetitive T, the lower is r). Let $z_1, z_2 \ldots z_r$ the lengths of the runs of 0s and $o_1, o_2 \ldots o_r$ those of the runs of 1s. They create arrays $Z = 10^{z_1-1}10^{z_2-1} \ldots$ and $O = 10^{o_1-1}10^{o_2-1} \ldots$, which have overall $2r$ 1s out of $2n$, and hence can be compressed to $2r \log \frac{n}{r} + O(r) + o(n)$ bits with constant-time *rank* and *select* [24].

While Sadakane [26] represented explicitly the suffix tree topology using $4n$ bits, Fischer et al. showed that all the operations can be simulated with suffix array ranges, by means of three operations on LCP: (1) $RMQ(i, j)$ gives the position of the minimum in $LCP[i, j]$; (2) $PSV(i)$ finds the last value smaller than $LCP[i]$ in $LCP[1, i-1]$; and (3) $NSV(i)$ finds the first value smaller than $LCP[i]$ in $LCP[i+1, n]$. All these could easily be solved in constant time using

$O(n)$ extra bits of space on top of the LCP representation, but Fischer et al. give sublogarithmic-time algorithms to solve them with only $o(n)$ extra bits.

Cánovas and Navarro [3] implemented a practical solution to solve the operations *NSV/PSV/RMQ*. They divided the LCP array into blocks of length L and formed a hierarchy of blocks, where they store the minimum LCP value of each block i in an array $m[i]$. The array uses $\frac{n}{L}\log n$ bits. On top of array m, they construct a perfect L-*ary* tree T_m where the leaves are the elements of m and each internal node stores the minimum of the values stored in its children. The total space needed for T_m is $\frac{n}{L}\log n(1+O(1/L))$ bits, so if $L=\omega(\log n)$, the space used is $o(n)$ bits. To answer the queries with this structure one computes a minimal cover in T_m of the range of interest of LCP and finds the node of the cover containing the answer. Then one moves down from the node until finding the right leaf of T_m. Finally, the corresponding LCP block is sequentially scanned to find the exact position, which is the heaviest part in practice. To answer RMQ queries faster they store for every node of T_m the local position in the children where the minimum occurs, so there in no need to scan the child blocks when going down the tree. The extra space incurred is still $o(n)$ bits. If the access to LCP cells is done via $PLCP$, then the overall cost of the operations is dominated by $O(L)$ times the cost of accessing a suffix array cell $A[i]$.

4 Re-Pair and Repetition-Aware CSAs

Re-Pair [15] is a grammar-based compression method that factors out repetitions in a sequence. This method is based on the following heuristic: (1) Find the most repeated pair ab in the sequence; (2) Replace all its occurrences by a new symbol s; (3) Add a rule $s \to ab$ to a dictionary R; (4) Iterate until every pair is unique.

The result of the compression of a text T over a alphabet Σ of size σ, is the dictionary R and the remaining sequence C, containing new symbols (s) and symbols in Σ. Every sub-sequence of C can be decompressed locally by the following procedure: Check if $C[i] < \sigma$; if so the symbol is original, else look in R for rule $C[i] \to ab$, and recursively continue expanding with the same steps.

The dictionary R corresponds to a context free grammar, and the sequence C to the initial symbols of the derivation tree that represents T. The final structure can be regarded as a sequence of binary trees with roots $C[i]$, see Figure 1 (left).

González and Navarro [10] used Re-Pair to compress the differentially encoded suffix array, $A'[i] = A[i] - A[i-1]$. They showed that Re-Pair achieves $|R| + |C| = O(r \log \frac{n}{r})$ on A', r being the number of *runs in* Ψ. A run in Ψ is a maximal contiguous area where $\Psi[i+1] = \Psi[i] + 1$. It was shown that the number of runs in Ψ is $r \le nH_k + \sigma^k$ for any k [16]. More importantly, repetitions in T induce long runs in Ψ, and hence a smaller r [17]. An exact bound has been elusive, but Mäkinen et al. [17] gave an average-case upper bound for r: if T is formed by a random base sequence of length $n' \ll n$ and then other sequences that have m random *mutations* (which include indels, replacements, block moves, etc.) with respect to the base sequence, then r is at most $n' + O(m \log_\sigma n)$ on average.

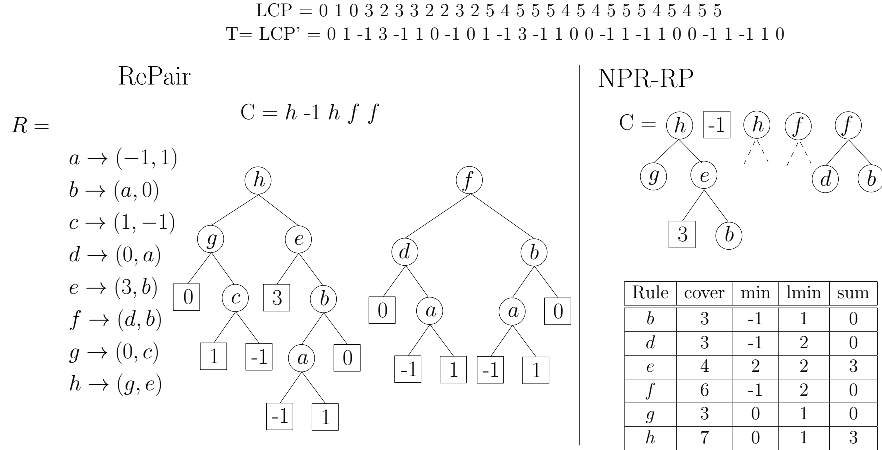

Fig. 1. On the left, example of the Re-Pair compression of a sequence T. We show R in array form and also in tree form. On the right, our NPR-RP construction over $LCP' = T$, pruning with $t = 4$. We show how deep can the symbols of C be expanded after the pruning.

The RLCSA [17] is a CSA where those runs in Ψ are factored out, to achieve $O(r)$ cells of space. More precisely, the size of the RLCSA is $r(2\log(n/r) + \log\sigma)(1+o(1))$ bits, where r is the number of runs in Ψ [17]. It supports accesses to A in time $O(s\log n)$, with $O((n/s)\log n)$ extra bits for a sampling of A.

5 Our Repetition-Aware CST

As explained in the Introduction, we use the RLCSA [17] as the base CSA of our repetition-aware CST. We also use the compressed representation of $PLCP$ [8]. Since in our case $r \ll n$, we use a compressed bitmap representation that is useful for very sparse bitmaps [13]: We δ-encode the runs of 0s between consecutive 1s, and store absolute pointers to the representation of every sth 1. This is very efficient in space and solves $select_1$ queries in time $O(s)$, which is the operation needed to compute a $PLCP$ value.

The main issue is how to support fast operations using the RLCSA and our LCP representation. As already explained, we choose to support all the operations using RMQ/PSV/NSV [8], and in turn follow the scheme of Cánovas and Navarro [3] to support these using the tree T_m. A problem is that this tree is of size $O((n/L)\log n)$ bits, insensitive to the repetitiveness of the text.

Our main idea is to replace the regular structure of tree T_m by the parsing tree obtained by a grammar compressor of the sequence LCP. We will now explain this idea in detail.

5.1 Grammar-Compressing the LCP Array

The following fact motivates grammar-compressing the LCP array.

Fact 1 *If $i-1$ and i are within a run of Ψ and $T[A[i-1]] = T[A[i]]$, then $LCP[i] = LCP[\Psi(i)] + 1$.*

Proof. Let $j = \Psi(i)$ and $j' = \Psi(i-1)$, and call $\ell = LCP[j]$. Then, if $x = T[A[i-1]] = T[A[i]]$, it holds $T[A[i-1], n] = x \cdot T[A[j'], n]$ and $T[A[i], n] = x \cdot T[A[j], n]$, thus $LCP[i] = \ell + 1$.

This means that, except for the σ places of $A[1, n]$ where the first characters of suffixes change, runs in Ψ correspond to areas of LCP that are off by 1 with respect to other areas of LCP. This is the same situation detected by González and Navarro [10] on $A[1, n]$. Following their idea, we can grammar-compress array LCP', defined as $LCP'[i] = LCP[i] - LCP[i-1]$ if $i > 1$, and $LCP'[1] = LCP[1]$. This differential LCP array contains now $O(r)$ areas that are exact repetitions of others, and a RePair-based compression of it yields $|R| + |C| = O(r \log \frac{n}{r})$ words [10]. We note, however, that the compression achieved in this way [10] is modest: we guarantee $O(r \log \frac{n}{r})$ *words*, whereas the RLCSA and PLCP representations require basically $O(r \log \frac{n}{r})$ *bits*. Thus we do not apply this idea directly, but rather *truncate* the parsing tree of the grammar, and use it as a device to speed up computations that would otherwise require expensive accesses to $PLCP$.

Let R and C be the results of compressing LCP' with RePair. Every nonterminal i of R expands to a substring $S[1, t]$ of LCP'. No matter where S appears in LCP' (indeed, it must appear more than once), we can store some values that are intrinsic to S. Let us define a relative sequence of values associated to S, as follows: $S'[0] = 0$ and $S'[j] = S[j] + S'[j-1]$. Then, we define the following variables associated to the nonterminal:

- $min_i = \min_{1 \le j \le t} S'[j]$ is the minimum value in S'.
- $lmin_i$ and $rmin_i$ are the leftmost and rightmost positions j where $S'[j] = min_i$, respectively.
- $sum_i = S'[t] = \sum_{1 \le j \le t} S[j]$ is the sum of the values $S[j]$.
- $cover_i = t$ is the number of values in S'.

As most of these values are small, we encode them with Directly Addressable Codes [2], which use less space for short numbers while providing fast access ($rmin$ stored as the difference with $lmin$).

To reduce space, we prune the grammar by deleting the nonterminals i such that $cover_i < t$, where t will be a space/time tradeoff parameter (recall that the grammar is superfluous, as we have access to LCP via $PLCP$, so we use it only to speed up computations). However, "short" nonterminals that are mentioned in sequence C are not deleted.

This ensures that we can skip $\Omega(t)$ symbols of LCP with a single access to the corresponding nonterminal in C, except for the short nonterminals (and terminals) that are retained in C. To speed up traversals on C, we join together maximal consecutive subsequences of nonterminals and terminals in C that sum

up a total $cover < t$: we create a new nonterminal rule in R (for which we precompute the variables above) and replace it in C, deleting those nonterminals that formed the new rule and do not appear anymore in C. This will also guarantee that no more than $O(t)$ accesses to LCP are needed to solve queries. Note that we could have built a hierarchy of new nonterminals by recursively grouping t consecutive symbols of C, achieving logarithmic operation times just as with tree T_m [3], but this turned out to be counterproductive in practice. Figure 1 (right) gives an example.

Finally, sampled pointers are stored to every c-th cell of C. Each sample for position $C[c \cdot j]$, stores:

- $Pos[j] = 1 + \sum_{1 \leq k \leq cj-1} cover_{C[k]}$, that is, the first position $LCP[i]$ corresponding to $C[c \cdot j]$.
- $Val[j] = \sum_{1 \leq k \leq cj-1} sum_{C[k]}$, that is, the value $LCP[i]$.

5.2 Computing NSV, PSV, and RMQ

To answer $NSV(i)$, we first look for the rule $C[j]$ that contains $LCP[i+1]$: we binary search Pos for the largest j' such that $Pos[j'] \leq i+1$ and then sequentially advance on $C[cj'..j]$ until finding the largest j such that $pos = Pos[j] + \sum_{cj' \leq k < j} cover_{C[k]} \leq i+1$. At the same time, we compute $\ell = Value[j'] + \sum_{cj' \leq k < j} sum_{C[k]}$.

Now, if $\ell + min_{C[j]} < LCP[i]$, it is possible that $NSV(i)$ is within the same $C[j]$. In this case, we search recursively the tree expansion with root $C[j]$ for the leftmost value to the right of i and smaller than $LCP[i]$: Let $C[j] \rightarrow ab$ in the grammar. We recursively visit child a if $\ell + min_a < LCP[i]$ and $pos + cover_a > i+1$. If we find no answer there, or we had decided not to visit a, then we set $\ell = \ell + sum_a$ and $pos = pos + cover_a$ and recursively visit child b if $\ell + min_b < LCP[i]$ and $pos + cover_b > i+1$. If we also find no answer inside b, or we had decided not to visit b, we return with no value. On the other hand, if we reach a leaf l during the recursion, we sequentially scan the array $LCP[pos, pos + cover_l - 1]$, updating $\ell = \ell + LCP[k]$ and increasing pos. If at some position we find a value smaller than $LCP[i]$, we report the position pos.

If we return with no value from the first recursive call at $C[j]$, it was because the only values smaller than $LCP[i]$ were to the left of i. In this case, or if we had decided not to enter into $C[j]$ because $\ell + min_{C[j]} \geq LCP[i]$, we sequentially scan $C[j, n]$, while updating $\ell = \ell + sum_{C[k]}$ and $pos = pos + cover_{C[k]}$, until finding the first k such that $\ell + min_{C[k]} < LCP[i]$. Once we find such k, we are sure that the answer is inside $C[k]$. Thus we enter into $C[k]$ with a procedure very similar to the one for $C[j]$ (albeit slightly simpler as we know that all the positions are larger than i). In this case, as the LCP values are discrete, we know that if $\ell + min_{C[k]} = LCP[i] - 1$, there is no smaller value to the left of the min value, so in this case we directly answer the corresponding $lmin$ value, without accessing the LCP array. The solution to $PSV(i)$ is symmetric.

To answer $RMQ(x, y)$, we find the rules $C[i]$ and $C[j]$ containing x and y, respectively. We sequentially scan $C[i+1, j-1]$ and store the smallest $\ell + min_{C[k]}$

value found (in case of ties, the leftmost). If the minimum is smaller than the corresponding values $\ell + min_{C[i]}$ and $\ell + min_{C[j]}$, we directly return the value $pos{+}lmin_{C[k]}$ corresponding to position $C[k]$. Else, if the global minimum in $C[i]$ is equal to or less than the minimum for $i < k < j$, we must examine $C[i]$ to find the smallest value to the right of $x - 1$. Assume $C[i] \rightarrow ab$. We recursively enter into a if $pos + cover_a > x$, otherwise we skip it. Then, we update $\ell = \ell + sum_a$ and $pos = pos + cover_a$, and enter into b if $pos < x$, otherwise we directly consider $\ell + min_b$ as a candidate for the minimum sought. Finally, if we arrive at a leaf we scan it, updating ℓ and pos, and consider all the values ℓ where $pos \geq x$ as candidates to the minimum. The minimum for $C[i]$ is the smallest among all candidates to minimum considered, and with $pos + lmin_b$ or the leaf scanning process we know its global position. This new minimum is compared with the minimum of $C[k]$ for $i < k < j$. Symmetrically, in case $k = j$ contains a value smaller than the minimum for $i \leq j < j$, we have to examine $C[j]$ for the smallest value to the left of $y + 1$.

6 Experimental Evaluation

We used various DNA collections from the Repetitive Corpus at *PizzaChili* (`http://pizzachili.dcc.uchile.cl/repcorpus`, created and thoroughly studied by Kreft [12]). We took DNA collections `Para` and `Influenza`, which are the most repetitive ones, and `Escherichia`, a less repetitive one. These are collections of genomes of various bacteria of those species. We also use `DNA`, which is plain DNA from *PizzaChili*, as a non-repetitive baseline. On the other hand, in order to show how results scale with repetitiveness, and although it is not a biological collection, we also included `Einstein`, corresponding to the Wikipedia versions of the article about Einstein in German.

All experimental results were performed on a 8 GB RAM computer with Intel Core2 Duo, each processor of 3 GHz, 6 MB cache. Our implementation will be publicly available at the ICDB Web page (`http://www.icdb.cl/software.html`).

For the RLCSA we used a fixed sampling that gave reasonable performance: one absolute value out of 32 is stored to access $\Psi(i)$, and one text position every 128 is sampled to compute $A[i]$. Similarly, we used sampling step 32 for the δ-encoding of the bitmaps Z and O that encode $PLCP$.

Table 1 shows the resulting sizes. The bpc of the CST is partitioned into those for the RLCSA, for the PLCP, and for NPR, which stands for the data structure that solves $NSV/PSV/RMQ$ queries. For the latter we used the smallest setting that offered answers within 2 milliseconds (msec). It can be seen that we obtain, overall, 1.3–1.5 bpc for the most repetitive DNA collections. This value deteriorates until approaching, for non-repetitive DNA, the same 10 bpc that are reported in the literature for existing CSTs. Thus our data structure adapts smoothly to the repetitiveness (or lack of it) of the collection. On the other hand, on Einstein, which is much more repetitive, the space gets as low as 0.28 bpc. This is a good indication of what we can expect on future databases

Name	Text MB	CST MB	RLCSA	(P)LCP	NPR	Total
Para	410	67	0.84	0.26	0.20	1.30
Influenza	148	27	0.96	0.21	0.30	1.47
Escherichia	108	48	2.46	0.92	0.20	3.58
DNA	50	61	5.91	3.62	0.30	9.83
Einstein	89	3	0.17	0.01	0.10	0.28

Table 1. Text sizes, size of our CST (which replaces the text), bpc for the different components, and total bpc of the different collections considered. The NPR structure is the smallest setting between NPR-RP and NPR-RPBal for that particular text.

with thousands of individuals of the same species, as opposed to these testbeds with a few tens of individuals, or with more genetic variation.

Let us discuss the NPR operations now. We used a public Re-Pair compressor by ourselves (`http://www.dcc.uchile.cl/gnavarro/software`), which offers two alternatives when dealing with symbols of the same frequencies. The basic one, that we will call NPR-RP, stacks the symbols, whereas the second one, NPR-RPBal, enqueues them and obtains much more balanced grammars in practice. For our structure we tested values $t = c = 64, 128, 256, 512$. We also include the basic regular structure of Cánovas and Navarro [3] (running over our RLCSA and PLCP representations), to show that our grammar-based version offers better space/time tradeoffs than their regular tree T_m. For this version, RP-CN, we used values $L = 36, 64, 128, 256, 512$.

We measure the times of operations NSV (as PSV is symmetric) and RMQ following the methodology of Cánovas and Navarro [3]. We choose 10,000 random suffix tree leaves (corresponding to uniformly random suffix array intervals $[v_l, v_r] = [v, v], v \in [1, n]$) and navigate towards the root using operation $parent(v_l, v_r) = [PSV(v_l), NSV(v_r)]$. At each such node, we also measure the *string depth*, which corresponds to query $strdep(v_l, v_r) = LCP[RMQ(v_l+1, v_r)]$. We average the times of all the NSV and RMQ queries performed.

Figure 2 shows the space/time performance of NPR-CN, NPR-RP, and NPR-RPBal. In addition, Figure 2 shows the number of explicit accesses to LCP made per NPR operation, showing that in practice the main cost of the NPR operations lies in retrieving the LCP values. Clearly, NPR-RP and NPR-RPBal dominate the space/time map for all queries. They always make better use of the space than the regular tree of NPR-CN. NPR-RPBal is usually better than NPR-RP, especially in RMQ queries, where NPR-RP suffers from extremely unbalanced trees that force the algorithm to examine many nodes, one by one. There are some particular cases, like NSV on `Escherichia`, where NPR-RP is the fastest.

References

1. Apostolico, A.: The myriad virtues of subword trees, pp. 85–96. Combinatorial Algorithms on Words. NATO ISI Series, Springer-Verlag (1985)
2. Brisaboa, N., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Proc. 16th SPIRE. pp. 122–130 (2009)

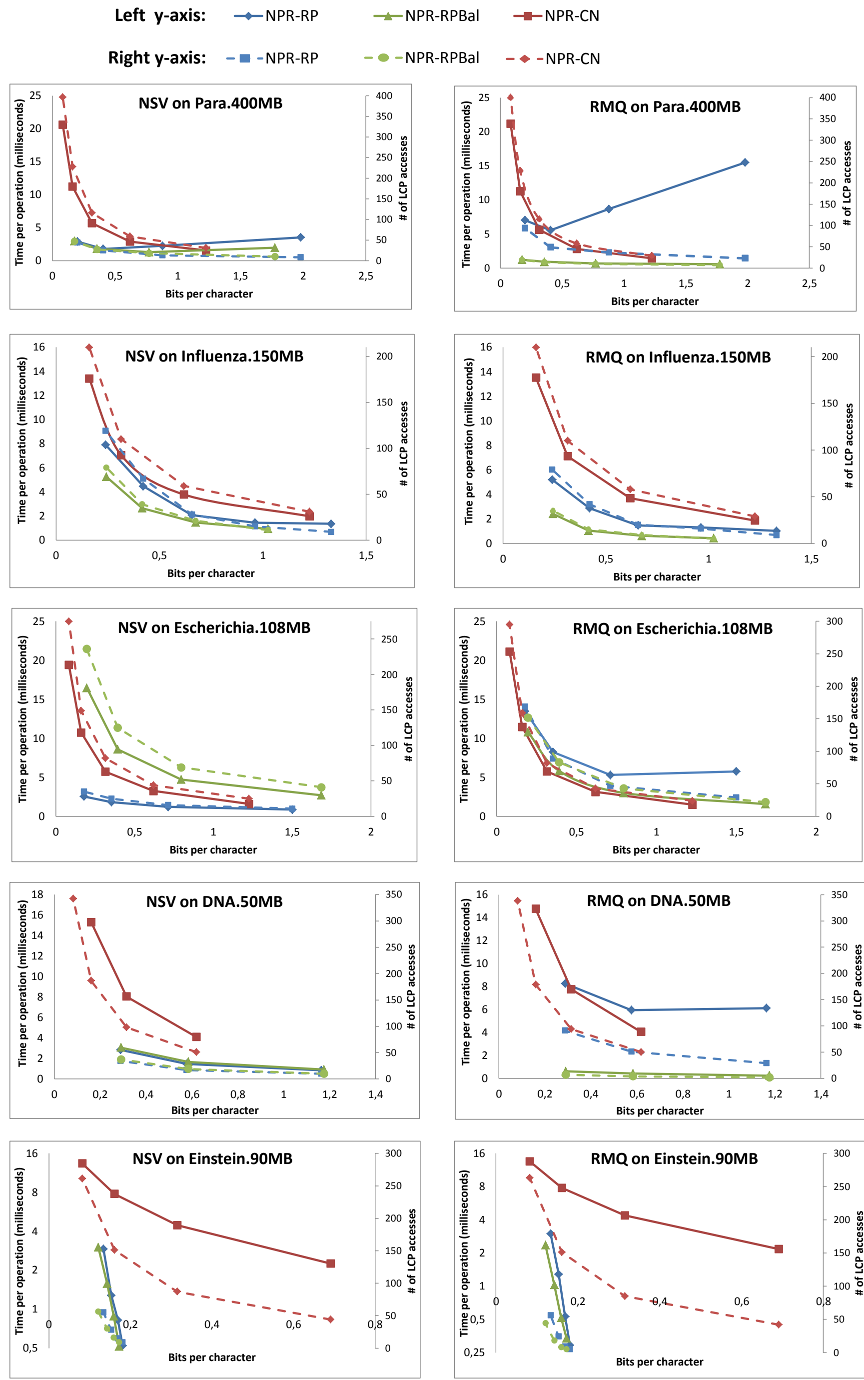

Fig. 2. Space/time performance of NPR operations. The x-axis shows the size of the NPR structure. Note the logscale on y for `Einstein`.

3. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Proc. 9th SEA. pp. 94–105 (2010)
4. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Compressed q-gram indexing for highly repetitive biological sequences. In: Proc. 10th BIBE. pp. 86–91 (2010)
5. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Proc. 34th MFCS. pp. 235–246 (2009)
6. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM J. Exp. Algor. 13, article 12 (2009)
7. Fischer, J.: Wee LCP. Inf. Proc. Lett. 110, 317–320 (2010)
8. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comp. Sci. 410(51), 5354–5364 (2009)
9. Gog, S.: Compressed Suffix Trees: Design, Construction, and Applications. Ph.D. thesis, Univ. of Ulm, Germany (2011)
10. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proc. 18th CPM. pp. 216–227 (2007)
11. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
12. Kreft, S.: Self-index based on LZ77. Master's thesis, Univ. of Chile (2010), arXiv:1112.4578v1
13. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Proc. 22nd CPM. pp. 41–54 (2011)
14. Kurtz, S.: Reducing the space requirements of suffix trees. Report 98-03, Technische Fakultät, Univ. Bielefeld, Germany (1998)
15. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
16. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. Comp. 12(1), 40–66 (2005)
17. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comp. Biol. 17(3), 281–308 (2010)
18. Manber, U., Myers, E.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. pp. 935–948 (1993)
19. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
20. McCreight, E.: A space-economical suffix tree construction algorithm. J. ACM 32(2), 262–272 (1976)
21. Munro, I.: Tables. In: Proc. 16th FSTTCS. pp. 37–42 (1996)
22. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
23. Ohlebusch, E., Fischer, J., Gog, S.: CST++. In: Proc. 17th SPIRE. pp. 322–333 (2010)
24. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA. pp. 233–242 (2002)
25. Russo, L., Navarro, G., Oliveira, A.: Fully-compressed suffix trees. ACM Trans. Alg. 7(4), article 53 (2011)
26. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comp. Sys. 41(4), 589–607 (2007)
27. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. Swit. and Aut. Theo. pp. 1–11 (1973)