Faster Compressed Suffix Trees for Repetitive Collections

Gonzalo Navarro, University of Chile
Alberto Ordóñez, University of A Coruña

Recent compressed suffix trees targeted to highly repetitive sequence collections reach excellent compression performance, but operation times are very high. We design a new suffix tree representation for this scenario that still achieves very low space usage, only slightly larger than the best previous one, but supports the operations orders of magnitude faster. Our suffix tree is still orders of magnitude slower than general-purpose compressed suffix trees, but these use several times more space when the collection is repetitive. Our main novelty is a practical grammar-compressed tree representation with full navigation functionality, which is useful in all applications where large trees with repetitive topology must be represented.

Additional Key Words and Phrases: Grammar Compressed Trees, Suffix Trees, Repetitive Sequence Collections, Bioinformatics

1. INTRODUCTION

Suffix trees [Weiner 1973; McCreight 1976; Ukkonen 1995] are a favorite data structure in stringology, with a large number of applications in bioinformatics [Apostolico 1985; Gusfield 1997; Ohlebusch 2013], thanks to their versatility. By means of a small set of query and traversal primitives (see Table I), suffix trees yield efficient solutions to many complex problems on pattern matching, pattern discovery, string comparisons, and others. The main problem of suffix trees is their space usage, which can easily reach 20 bytes per text symbol. On DNA sequences, where each base can be represented in 2 bits, the suffix tree takes up to 80 times the text size!

A solution to the space problem could be to deploy the suffix trees on secondary memory [Ferragina and Grossi 1999; Crauser and Ferragina 2002; Kärkkäinen and Rao 2003; Dementiev et al. 2008; Ferragina et al. 2012; Kärkkäinen and Kempa 2014a; 2014b; Gog et al. 2014]. Unfortunately, most of the complex tasks carried out on suffix trees need to traverse them across arbitrary access paths, in which case secondary memory representations perform poorly due to the low locality of reference. The fact that suffix trees use much space but need to fit in main memory to operate efficiently restricts their applicability to small sequence collections only; for example, handling just one human genome requires a machine with 60GB of RAM.

Funded with basal funds FB0001, Conicyt, Chile, by Fondecyt Grant 1-140796, Chile, by CDTI EXP 000645663/ITC-20133062 (CDTI, MEC, and AGI), Xunta de Galicia (PGE and FEDER) ref. GRC2013/053, and by MICINN (PGE and FEDER) refs. TIN2009-14560-C03-02, TIN2010-21246-C02-01, TIN2013-46238-C4-3-R and TIN2013-47090-C3-3-PandAP2010-6038 (FPU Program).
A preliminary partial version of this paper appeared in *Proc. SEA 2014*.
Author's addresses: Gonzalo Navarro, Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, Chile. gnavarro@dcc.uchile.cl. Alberto Ordóñez, Laboratorio de Bases de Datos, Departamento de Computación, University of A Coruña, A Coruña, Spain. alberto.ordonez@udc.es.

DOI:http://dx.doi.org/10.1145/0000000.0000000

A number of engineered representations of suffix trees have been proposed to cope with their space problem [Kurtz 1999; Abouelhoda et al. 2004], but these still take 6 to 10 bytes per symbol. Suffix arrays [Manber and Myers 1993] reduce the space to about 4 bytes per symbol, but they lose a number of suffix tree functionalities that are essential in many complex tasks (e.g., suffix links).

The emergence of compressed suffix arrays (CSAs) [Navarro and Mäkinen 2007], which managed to represent *both* the sequence and its suffix array within the space of the *compressed* sequence, paved the way to radical improvements in the area, by making it possible to build compressed suffix trees (CSTs) on top of CSAs. Sadakane [2007] introduced the first CST representation, which included a succinct representation of the tree topology and retained full suffix tree functionality. A recent, well engineered implementation by Gog [2015], requires about 10 bits per symbol (bps), that is, slightly more than one byte per symbol, on general DNA text (this includes the storage of the CSA, and thus of the sequence itself), and can perform all the operations in a few microseconds; the easy ones may need just a few nanoseconds. Fischer et al. [2009] and Fischer [2010] developed a new CST using even less space. An efficient variant by Ohlebusch et al. [2010] was shown to use about 8 bps [Gog 2015]. Their main idea was to avoid the explicit representation of the tree topology. Their operation times, as a consequence, are higher than Sadakane's, but still within microseconds. Russo et al. [2011] introduced an even smaller CST, using about 4 bps, yet raising operation times to milliseconds.

All these CSTs use space proportional to the *empirical entropy* of the text collection [Manzini 2001], which is a measure of statistical compressibility. In most DNA collections, however, the empirical entropy is also close to 2 bps, that is, DNA is essentially incompressible with statistical compressors[1]. Still, CSTs operating within microseconds can be built on a human genome (of about 3 billion bases), for example, and maintained in a main memory of about 3–4GB.

However, the goal of maintaining *one* human genome in main memory has quickly become outdated. The rapid improvements in sequencing technology have driven the growth of large genome repositories. Modern challenges are to handle repositories of thousand genomes (e.g., see the 1000-Genomes project, `http://www.1000genomes.org`). Further, one would like to efficiently perform complex bioinformatic analyses on those huge sequence collections, ideally maintaining a suffix tree on them. Even a CST using 1 byte per symbol is problematic when a thousand genomes must be maintained: we would need 3TB of main memory!

Fortunately, those fast-growing DNA collections are formed by the sequenced genomes of hundreds or thousands of individuals of the same species. This makes those collections *highly repetitive*; for example, an accepted figure is that two human genomes share 99.5% to 99.9% of their sequences [Jorde and Wooding 2004; Tishkoff and Kidd 2004].[2] Statistical compression does not take proper advantage of repetitiveness [Kreft and Navarro 2013], but other techniques like grammar or Lempel-Ziv compression do [Navarro 2012].

There have been some indexes aimed at performing pattern matching (i.e., just simple string searches) on repetitive collections based on those techniques [Claude and Navarro 2010; Kuruppu et al. 2011; Kreft and Navarro 2013; Claude and Navarro 2012; Do et al. 2012; Gagie et al. 2012]. However, they do not provide the versatile suffix tree functionality, and they do not seem to yield a way to obtain it. Instead, the so-called run-length CSA [Mäkinen et al. 2010] (RLCSA), although based in principle

[1]See, for example, `http://pizzachili.dcc.uchile.cl/texts.html`.
[2]This number may be even higher on individuals of the same geographic area, for example. There is always controversy about this number and on how it is measured, however.

on weaker compression techniques, yields a data structure that is useful to achieve CSTs for repetitive collections.

Building on the RLCSA and on the CST of Fischer et al. [2009], Abeliuk and Navarro [2012] (see also Abeliuk et al. [2013]) introduced the first CST for repetitive collections, by using grammar-compressed representations of some of their internal components. On the repetitive biological collections they tested, their CST used around 1–2 bps, well below the spaces achieved with the general-purpose CSTs. Their operation time was, however, in the order of milliseconds, which makes the structure far less attractive.

In this paper we introduce a new CST called GCT, for "grammar-compressed topology", that achieves low space on repetitive collections and much better times. The GCT operates in the order of microseconds, becoming much closer to the times of general-purpose CSTs [Sadakane 2007; Ohlebusch et al. 2010; Gog 2015], and actually outperforming the smallest members of that family [Russo et al. 2011; Cánovas and Navarro 2010; Abeliuk et al. 2013] (which are still significantly larger than the GCT on repetitive collections). On synthetic DNA collections with 99.9% similarity, our GCT uses 2 bps, whereas the previous CST for repetitive collections uses 1.5 bps, and their difference shrinks as the collections become more repetitive. In exchange for this higher space, the GCT is up to 3 orders of magnitude faster.

To achieve this result, we build on the CST of Sadakane [2007], but use grammar compression on the tree topology, instead of just a succinct representation. More precisely, we use string grammar compression on the sequence of parentheses that represents the suffix tree topology (an idea briefly sketched by Bille et al. [2011] for arbitrary trees). A repetitive text collection turns out to have a suffix tree with repetitive topology, and having the tree represented in this form allows us to speed up many operations that are very slow to simulate without the explicit topology [Fischer et al. 2009; Russo et al. 2011].

The GCT retains the full functionality of succinct tree representations [Navarro and Sadakane 2014], but is likely to use much less space when the tree has frequent repeated substructures. While we do not prove worst-case results on the GCT representation, our experiments show that it performs well in the scenario studied in this paper. The GCT is likely to be useful in other applications where a tree with repetitive topology must be represented; we describe a couple of them in the Conclusions. The theoretical proposal of Bille et al. [2011] does offer some guarantees, but we believe it would not perform so well in practice due to some space overheads incurred to perform well in the worst case.

2. BASIC CONCEPTS

2.1. Succinct Tree Representations

Among the many succinct tree representations, we describe the one of Navarro and Sadakane [2014], on which we build. We choose this representation because it implements a large number of tree operations on top of a simple representation. The tree topology is represented using a sequence of parentheses. We traverse the suffix tree in preorder, writing an opening parenthesis when we first arrive at a node, and a closing one when we leave its subtree. Thus a tree of t nodes is represented with $2t$ parentheses, as a binary sequence $P[1, 2t]$. Each node is identified with the offset of its opening parenthesis in P, so we can speak of "node i" to refer to the one represented by $P[i] = ' ('$.

We define the *excess* of a position, $E(i)$, as the number of opening minus closing parentheses in $P[1, i]$. Note that $E(i)$ is the depth of node i. Many tree navigation operations can be carried out with two operations related to the excess: *fwd*(i, d) is the smallest $j > i$ such that $E(j) = E(i) - d$, and *bwd*(i, d) is the largest $j < i$ such that

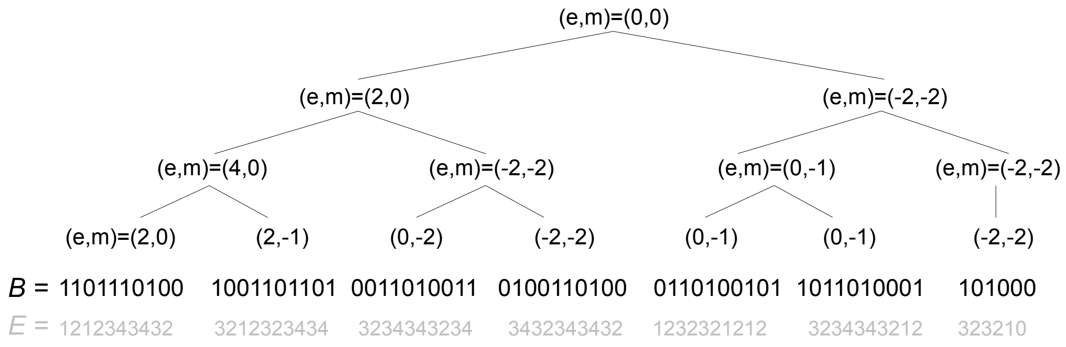

(e,m)=(0,0)
(e,m)=(2,0)
(e,m)=(-2,-2)
(e,m)=(4,0)
(e,m)=(-2,-2)
(e,m)=(0,-1)
(e,m)=(-2,-2)
(e,m)=(2,0)
(2,-1)
(0,-2)
(-2,-2)
(0,-1)
(0,-1)
(-2,-2)
B = 1101110100 1001101101 0011010011 0100110100 0110100101 1011010001 101000
E = 1212343432 3212323434 3234343234 3432343432 1232321212 3234343212 323210

Fig. 1: The data structure for the succinct representation of a parentheses sequence $P[1, 2t]$ (which corresponds to the topology of the suffix tree in Figure 2). It is formed by bitvector $B[1, 2t]$, which represents P with 1-bits for the '('s and 0-bits for the ')'s, and the tree of block summaries on top of it. We show in gray the values $E(i)$, which are not represented explicitly.

$E(j) = E(i) - d$. For example, the parenthesis closing the one that opens at position i is at $\mathit{fwd}(i, 1)$, so the next sibling of node i is $j = \mathit{fwd}(i, 1) + 1$ if $P[j] =' ('$, else i is the last child of its parent. Analogously, the previous sibling is $\mathit{bwd}(i-1, 0)+1$ if $P[i-1] =')'$, else i is the first child of its parent. A node i is a leaf if $P[i + 1] =')'$, otherwise its first child is $i+1$. The number of nodes in the subtree rooted at i is $(\mathit{fwd}(i, 1)-i+1)/2$. Node i is an ancestor of j if $i \leq j \leq \mathit{fwd}(i, 1)$. The parent of node i is $\mathit{bwd}(i, 2) + 1$ and the h-th level ancestor is $\mathit{bwd}(i, h + 1) + 1$. The preorder value of a node, $\mathit{preorder}(i)$, is the number of opening parentheses in $P[1, i]$; note that $\mathit{preorder}(i) = (E(i) + i)/2$. The inverse of *preorder* is $\mathit{node}(j)$, which gives the node with preorder j and is solved analogously to *fwd*, this time looking for a certain value of $i + E(i)$. A more complex operation is to find the lowest common ancestor of two nodes, $\mathit{LCA}(i, j)$. Unless one is the ancestor of the other, computing *LCA* requires operation *RMQ* (range minimum query) on the virtual array of depths: $\mathit{RMQ}(i, j)$ is the position of a minimum in $E(i)E(i+1) \dots E(j)$, and then $\mathit{LCA}(i, j) = \mathit{parent}(\mathit{RMQ}(i, j) + 1)$. Many other operations are available with the primitives E, *fwd*, *bwd*, and *RMQ* [Navarro and Sadakane 2014].

To implement those primitives, the sequence $P[1, 2t]$ is cut into blocks of $b \log t$ parentheses, for a parameter b (we use base 2 logarithms by default). For each block k we store $m[k]$, the minimum excess within the block, and $e[k]$, the total excess within the block. The blocks are the leaves of a perfect binary tree of higher-level blocks, for which we also store $m[k]$ and $e[k]$. See Figure 1 for an example.

In this representation, operation $\mathit{fwd}(i, d)$ can be solved in $O(b+\log t)$ time as follows. Let k be the block where position i belongs. First, we scan P from $i + 1$ to the end of the block, to see if the desired excess difference is reached within the block. The block can be scanned by chunks of $(\log t)/2$ parentheses by using global precomputed tables of just $\sqrt{t}$ entries, which store the total and minimum excess in every possible chunk. If the answer is not inside the block, let d' be d plus the accumulated excess between $i+1$ and the end of the block; then d' is the new excess difference sought to the right of block k (recall that we seek for the smallest $j > i$ such that $E(j) = E(i) - d'$). Now we move to the parent of block k in the balanced tree. If k is the left child of its parent and k' is the right sibling of k, then if $d' > -m[k']$, we know that the desired excess is not reached within block k', thus we set $d' \leftarrow d' + e[k']$ and continue recursively with the parent node of block k. If, instead, k is the right child of its parent, we simply continue recursively with its parent.

This upward traversal continues until we find a right sibling k' for which $d' \leq -m[k']$, thus the desired excess difference is reached within block k'. Now we start a downward traversal. We check whether the difference is reached inside the left child k'' of k': if $d' \leq -m[k'']$, then we descend to k''; otherwise we set $d' \leftarrow d' + e[k'']$ and descend to the right child of k'. When we finally arrive at a leaf block, we complete the operation *fwd*(i, d) by scanning its parentheses from the beginning of the block until we reach excess difference d'.

Operations *bwd* and ***RMQ*** are solved analogously, and computing $E(i)$ is simpler; see Navarro and Sadakane [2014] for more details[3]. By using, for example, $b = \Theta(\log t)$, one obtains $O(\log t)$ time for all the operations and $2t+o(t)$ bits to store the the parentheses plus the balanced tree of $m[]$ and $e[]$ values.[4]

2.2. Compressed Suffix Trees

Let $T[1, n]$ be a text (or the concatenation of the texts in a collection) over alphabet $\Sigma = [1, \sigma]$. The character at position i of T is denoted $T[i]$, whereas $T[i, j]$ denotes $T[i]T[i+1]\ldots T[j]$, a substring of T. A *suffix* of T is a substring of the form $T[i, n]$ and a *prefix* of T is of the form $T[1, i]$. The *suffix trie* of T is the digital tree formed by inserting all the suffixes of T, so that any substring of T labels the path from the root to a node of the suffix trie. In particular, any suffix $T[i, n]$ labels the path from the root to a leaf of the suffix trie; such leaf is marked with the position i. We consider that the character labels in the suffix trie are on the edges. For each tree node v, we call *str*(v) the string obtained by concatenating the labels on the edges between the root and v. The *suffix tree* of T [Weiner 1973] (see Figure 2 for an example) is formed by compressing the unary paths of the suffix trie into a unique edge, labeled with the concatenation of the labels of the compressed edges. The first characters of the (string) labels of the edges that lead to the children of any node are distinct, and we assume they are sorted by increasing value left to right. Table I lists the operations we aim to support on suffix trees.

The *suffix array* [Manber and Myers 1993] of T is an array $A[1, n]$ of values in $[1, n]$, formed by collecting the position marks of the suffix tree in left-to-right order. Alternatively, $A[1, n]$ can be seen as the array of all the suffixes of T sorted in lexicographic order.

2.2.1. Compressed suffix arrays. There are many compressed suffix arrays (CSAs) in the literature [Navarro and Mäkinen 2007]. The basic functionality they offer is (a) given a pattern $p[1, m]$, find the suffix array interval $A[sp, ep]$ of the suffixes of T that start with p (therefore $A[sp], A[sp+1], \ldots, A[ep]$ is the list of occurrences of p in T), (b) given a suffix array position i, return $A[i]$, (c) given a text position j, return $A^{-1}[j]$, that is, the position in A that points to the suffix $T[j, n]$, and (d) given $[l, r]$, obtain the text substring $T[l, r]$. Most CSAs achieve times of the form $O(m)$ to $O(m \log n)$ for operation (a), $O(\text{polylog}\, n)$ for (b) and (c), and at most $O((l-r)\log\sigma + \text{polylog}\, n)$ for (d). They require space $O(n \log \sigma)$ *bits* (as opposed to $O(n \log n)$ of classical suffix arrays), and in most cases close to the empirical entropy of T [Manzini 2001] (a measure of compressibility with statistical compressors). Note that, within this space, CSAs can reproduce any substring of T, so T does not need to be stored separately.

When T is *repetitive* (i.e., it can be represented as the concatenation of a few different substrings), then grammar and Lempel-Ziv compression greatly reduce its

[3]They describe a variant where the $e[]$ and $m[]$ values are absolute, not relative to the block; our description here is more similar to their dynamic variant. Also, they store a few more values to support other operations not usually required on suffix trees, and thus not considered in this paper.

[4]The theoretical proposal of Navarro and Sadakane [2014] obtains constant times, but the practical implementation [Arroyuelo et al. 2010] reaches logarithmic times.

Operation	Description
root()	the root node of the tree
tDepth(v)	the tree depth of node v
fChild(v)	the alphabetically first child of v
nSibling(v)	the alphabetically next sibling of v
pSibling(v)	the alphabetically previous sibling of v
isLeaf(v)	true if v is a leaf node
ancestor(v,u)	true if v is an ancestor of u
subtree(v)	number of nodes in the subtree rooted at v
parent(v)	the parent of v
tAncestor(v,d)	the ancestor of v at depth d
preorder(v)	the preorder number of v
LCA(v,u)	lowest common ancestor of nodes v and u
sDepth(v)	$\|$*str*$(v)\|$
letter(v,i)	*str*$(v)[i]$
child(v,a)	u such that $a \in \Sigma$ is the first letter on edge (v,u)
sLink(v)	u such that *str*$(u) = \beta$ in case *str*$(v) = a\beta$ and $a \in \Sigma$
sAncestor(v,d)	the highest ancestor u of v such that $\|$*str*$(u)\| \geq d$
textPos(v)	i such that *str*(v) starts at $T[i]$ (for a leaf v)

Table I: Typical operations supported by a suffix tree. The first group is composed of generic tree operations, whereas the second is specific of suffix trees. By *str*(v) we denote the string obtained by concatenating the labels on the edges between the root and v.

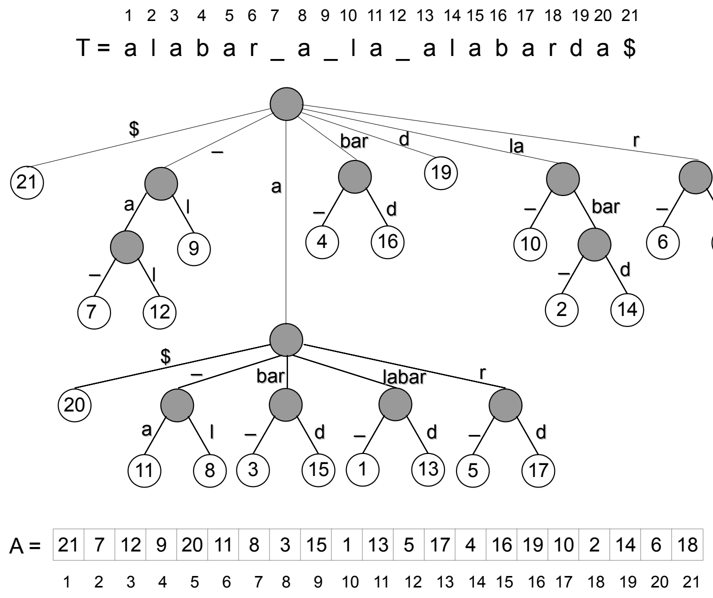


Fig. 2: Suffix tree and suffix array A for the text example $T = alabar\_a\_la\_alabarda\$$.

space. Instead, statistical compression does not profit from repetitiveness [Kreft and

Navarro 2013], and thus classical CSAs do not compress well repetitive text collections. Mäkinen et al. [2010] introduced the *run-length CSA (RLCSA)*, which compresses better when T is repetitive.

2.2.2. Longest common prefix array. The *longest common prefix (LCP)* array, *LCP*$[1, n]$, is a key component of various suffix tree representations. It stores in *LCP*$[i]$ the length of the longest common prefix between the suffixes $T[A[i], n]$ and $T[A[i-1], n]$ (with *LCP*$[1] = 0$).

Sadakane [2007] showed how to represent *LCP* using just $2n$ bits, by representing *PLCP*$[1, n]$ instead, where *PLCP*$[j]$ $=$ *LCP*$[A^{-1}[j]]$ (or *LCP*$[i]$ $=$ *PLCP*$[A[i]]$), that is, *PLCP* is *LCP* represented in text order, not in suffix array order. The key property is that *PLCP*$[j+1] \geq$ *PLCP*$[j] - 1$, which allows *PLCP* be represented using a bitvector $H[1, 2n]$, at the price of having to compute $A[i]$ in order to compute *LCP*$[i]$.

Fischer et al. [2009] proved that H was in addition compressible when the text was statistically compressible, but Cánovas and Navarro [2010] found out that the compression was not significant on standard texts. Instead, Abeliuk and Navarro [2012] showed that the technique proposed to compress H [Fischer et al. 2009] worked very well on repetitive texts.

2.2.3. Current compressed suffix trees. Sadakane [2007] showed that a functional compressed suffix tree (CST), that is, a data structure that solves the operations in Table I, could be represented with three components: (1) a compressed suffix array (CSA), (2) a compressed LCP array, and (3) a compressed representation of the topology of the suffix tree. Thus, other elements, like the string labels, were computed from these components without representing them.

Sadakane used an existing CSA, compressed the LCP array to $2n$ bits as described in Section 2.2.2, and represented the tree topology using succinct trees, which take $2n$ to $4n$ bits since the suffix tree has $t = n$ to $2n$ nodes. A study of such succinct tree representations [Arroyuelo et al. 2010] shows that the one described in Section 2.1 is well suited for the operations required on a suffix tree. Gog [2015] implemented Sadakane's CST, obtaining extremely fast operations.

Fischer et al. [2009] showed that one can operate without explicitly representing the tree topology, because each suffix tree node corresponds to a distinct suffix array interval. One can operate directly on those intervals, and all the tree operations can be simulated with three primitives on the intervals: *RMQ*(i, j) finds the (leftmost) position of the smallest value in *LCP*$[i, j]$, and *PSV*/*NSV*(i) finds the position in *LCP* preceding/following i with a value smaller than *LCP*$[i]$. Cánovas and Navarro [2010] implemented this theoretical proposal, speeding up the operations *RMQ* and *PSV*/*NSV* by building the balanced tree described in Section 2.1 on top of the *LCP* array (instead of on array E) and using ideas similar to those used to navigate trees [Navarro and Sadakane 2014] (albeit the application is quite different). Ohlebusch et al. [2010] presented a fast alternative implementation that uses $3n$ bits of space.

Abeliuk and Navarro [2012] proposed the first CST for repetitive text collections. They built on the representation of Cánovas and Navarro [2010], using the RLCSA and the compressed version of H to represent *LCP*, which, as mentioned, became compressible on repetitive texts. The only obstacle was that the balanced tree used to speed up *RMQ* and *PSV*/*NSV* operations was insensitive to repetitiveness. They overcame this by using the fact that the differential *LCP* array (*LCP*$[i]$ $-$ *LCP*$[i-1]$) is grammar-compressible, particularly on repetitive text collections. They applied RePair compression [Larsson and Moffat 2000] to the differential *LCP* array and used the grammar tree (whose nodes are the grammar nonterminals) instead of the incompressible balanced tree. That is, they stored the information needed to compute *PSV*/*NSV*/*RMQ* in the nodes of the grammar tree. As a result, they obtain very low space usage on

B = 1101110100100110110100110100110100110100011010010110110100011010000

$$\left(R = \begin{array}{l} R_0 \rightarrow 0 \\ R_1 \rightarrow 1 \\ R_2 \rightarrow R_1R_0 \\ R_3 \rightarrow R_1R_2 \\ R_4 \rightarrow R_2R_0 \\ R_5 \rightarrow R_3R_4 \\ R_6 \rightarrow R_5R_0 \\ R_7 \rightarrow R_5R_5 \end{array},\quad C = R_3\, R_1\, R_5\, R_4\, R_3\, R_7\, R_5\, R_6\, R_5\, R_2\, R_3\, R_6\, R_6 \right)$$

B =	1101	1	10100	100	110	110100110100	110100	1101000	110100	10	110	1101000	1101000
C =	R_3	R_1	R_5	R_4	R_3	R_7	R_5	R_6	R_5	R_2	R_3	R_6	R_6

Fig. 3: (R, C) is the result of applying RePair to the binary string B of Figure 1.

repetitive texts (from 0.6 to 4 bps, depending on the repetitiveness of the real-life collections used). A drawback is that the operations require milliseconds, instead of the microseconds required by most CSTs designed for standard text collections [Abeliuk et al. 2013].

2.3. Grammar Compression of Strings and Trees

Grammar compression of a string S is the task of finding a (context-free) grammar G that generates (only) S. RePair [Larsson and Moffat 2000] is a compression algorithm that finds such a grammar in linear time and space. It finds the most frequent pair ab of characters in S, creates a rule $X \rightarrow ab$, replaces all ab in S by X, and iterates until the most frequent pair appears only once (in subsequent iterations, a and/or b may be nonterminals). The final product of RePair is a set R of rules of the general form $X \rightarrow YZ$ and a sequence C of terminals and nonterminals corresponding to the final reduced version of S after all the replacements. For uniformity, we will add an initial nonterminal $X \rightarrow a$ for each terminal a, and in all the rules $X \rightarrow YZ$, Y and Z will always be nonterminals (this is called Chomsky normal form). Similarly, C will be formed only by nonterminals. Figure 3 shows an example.

To retrieve the string $\mathit{str}(X)$ represented by a nonterminal symbol X, we proceed recursively: If $X \rightarrow YZ$, then $\mathit{str}(X) = \mathit{str}(Y) \circ \mathit{str}(Z)$, where $\circ$ denotes string concatenation. The recursion ends when we arrive at terminals. We can thus expand any nonterminal X in optimal time $O(|\mathit{str}(X)|)$.

Grammar compression can also be applied to trees, by using grammars that generate trees instead of strings [Comon et al. 2007]. The simplest grammar is one that replaces full trees, so the associated grammar compression seeks for the minimal DAG (directed acyclic graph) equivalent to the tree. More powerful variants allow nonterminals with variables, with which grammar compression can replace connected subgraphs of the tree [Maneth and Busatto 2004; Lohrey et al. 2011]. In general, supporting even the most basic traversal operations on those compressed trees is not trivial, even in the simplest DAG compression. Bille et al. [2011] sketch a simple idea that retains all the full power of navigational operations of succinct trees (see Section 2.1). They basically propose to grammar-compress the string of parentheses $P[1, 2t]$ that represents the tree, attaching $m[]$ and $e[]$ values (and others, as needed) to the nonterminals in order to support efficient navigation. They prove that this compression is as powerful as the simple DAG tree compression, provided some small fixes are applied to the grammar.

Note that this theoretical idea is what was implemented in practice by Abeliuk and Navarro [2012], as described in Section 2.2.3, for solving queries on the *LCP* array:

using the RePair grammar tree instead of a balanced tree for storing $m[]$ and $e[]$ information. In this paper we implement the idea on the excess array of an actual tree — the suffix tree of the text. Unlike Bille et al., we do not alter the grammar given by RePair, but use it directly.

3. A NEW CST FOR REPETITIVE TEXT COLLECTIONS

We introduce a new CST tailored to repetitive texts, building on the original proposal of Sadakane [2007]. We use the RLCSA as the suffix array, and the compressed representation of H [Fischer et al. 2009; Abeliuk and Navarro 2012] for the *LCP* array. Unlike the previous CST of Abeliuk and Navarro, we do represent the suffix tree topology, to avoid the large performance penalty of omitting it. As anticipated, this tree topology will be grammar-compressed to exploit repetitiveness. As a result, our CST will use slightly more space than that of Abeliuk and Navarro, but it will be orders of magnitude faster. We call it GCT, for "grammar-compressed topology". The rest of the section is devoted to describing the GCT.

3.1. GCT Structure

Let $R[1,r]$ be the rules (including initial rules generating the terminals '(' and ')') and $C[1,c]$ the final sequence resulting from applying RePair compression to the parentheses sequence $P[1,2t]$ (recall Figure 3). We use a version of RePair that yields balanced grammars (i.e., of height $O(\log t)$) in most cases.[5] We describe how we store R and C.

3.1.1. Storing the rules R. A plain storage of the rules $R[1,r]$ requires $2r\log r$ bits, as a simple array $R[k]=(i,j)$ meaning $R[k]\rightarrow R[i]R[j]$. Instead, we will simplify a technique described by Tabei et al. [2013], which uses only $r\log r + O(r)$ bits and permits extracting the right hand of any rule in time $O(\log r)$.

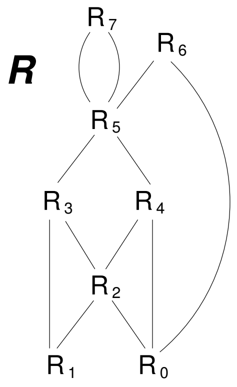
The grammar is seen as a DAG where the nodes are the nonterminals, and each rule $R_k \rightarrow R_iR_j$ induces two arrows, from R_k to R_i and from R_k to R_j. Now all the arrows from nodes to their left children, seen backward, form a tree T_L, and those to their right children, seen backward, form a tree T_R. We represent T_L and T_R, using a succinct tree representation, in $O(r)$ bits (recall Section 2.1). The identifiers of the nonterminals will be their preorder values in T_L: rule R_k will refer to the node with preorder k in T_L.

In addition, we need to map between a preorder value of a nonterminal in T_L and the preorder value of the same nonterminal in T_R, and back. We use a practical technique by Munro et al. [2003] that represents a permutation π of $\{1,2,\ldots,r\}$ using $r\log r+O(r)$ bits. It stores the array $\pi=[\pi(1),\pi(2),\ldots,\pi(r)]$ explicitly (thus $\pi(i)$ is computed in constant time), and adds $O(r)$ bits of data that allows computing any $\pi^{-1}(i)$ in time $O(\log r)$. Figure 4 illustrates this representation.

The main operation carried out on this representation is, given a nonterminal k such that $R_k \rightarrow R_iR_j$, find i and j. The procedure is as follows:

(1) Compute $x_L \leftarrow node(T_L,k)$, the node of T_L that represents R_k.
(2) Find $y_L \leftarrow parent(T_L,x_L)$, the parent of x_L, which represents R_i in T_L.
(3) Compute $i \leftarrow preorder(T_L,y_L)$, the identifier of nonterminal R_i.
(4) Map $k_R \leftarrow \pi(k)$, the preorder of R_k in T_R.
(5) Compute $x_R \leftarrow node(T_R,k_R)$, the node of T_R that represents R_k.
(6) Find $z_R \leftarrow parent(T_R,x_R)$, the parent of x_R, which represents R_j in T_R.
(7) Compute $j_R \leftarrow preorder(T_R,z_R)$, the preorder of z_R in T_R.
(8) Map back $j \leftarrow \pi^{-1}(j_R)$, the identifier of nonterminal R_k.

[5]From www.dcc.uchile.cl/gnavarro/software. There exist algorithms that ensure balanced grammars [Sakamoto 2005], but they are more complicated.



	m	e	s	l	pl	pr	str
1	0	1	1	0	(	(	1
2	0	1	3	1	(	)	110
3	0	0	6	2	(	)	110100
4	0	0	12	4	(	)	110100110100
5	−1	−1	7	2	(	)	1101000
6	0	0	2	1	(	)	10
7	−1	−1	3	1	(	)	100
8	−1	−1	1	0	)	)	0

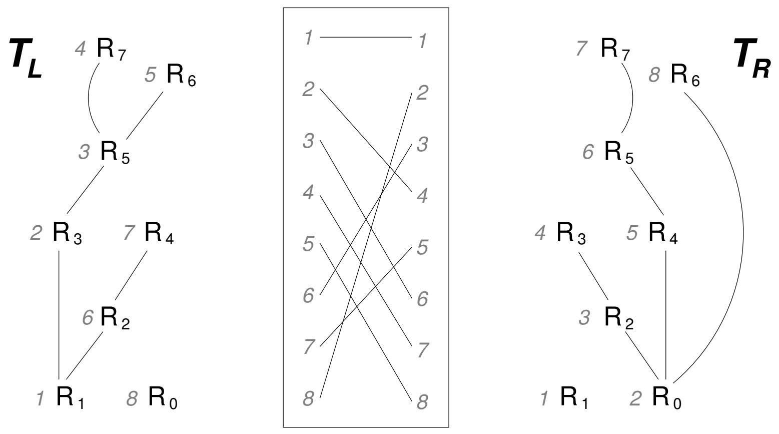


Fig. 4: The grammar R of Figure 3 in DAG form (top left), its representation as two trees T_L and T_R (with preorders in slanted gray) plus a permutation mapping preorders (bottom), and the data we store on the nonterminals (top right, node identifiers correspond to preorders in T_L).

In practice, the structure described in Section 2.1 is too powerful for the few operations we need on T_L and T_R. We use instead the so-called LOUDS representation [Jacobson 1989], which supports operation *parent* and an equivalent to operations *preorder* and its inverse *node* (that is, it assigns a distinct number in $[1, r]$ to each node, although it is not its preorder value). The LOUDS representation is smaller and faster than the one described in Section 2.1 [Arroyuelo et al. 2010], albeit it supports fewer operations.

3.1.2. Storing information on the rules. We enrich the grammar R with additional information on the nonterminals, to allow for fast operations on the represented tree. For each nonterminal R_k, we store the following arrays (see Figure 4).

(1) $m[k]$, the minimum excess of '('s in *str*(R_k). It holds $m[k] \leq 0$ because the excess of the empty prefix of the string is always 0.
(2) $e[k]$, the total excess of *str*(R_k).
(3) $s[k] = |$*str*$(R_k)|$, the length of the string R_k generates.

(4) $l[k]$, the number of leaf nodes represented inside $\mathit{str}(R_k)$, that is, the number of substrings of the form '()' (or '10', in bits) present in $\mathit{str}(R_k)$.
(5) $pl[k]$ and $pr[k]$, the leftmost and rightmost parentheses (bits) in $\mathit{str}(R_k)$.

Since $e[k]$ can be positive or negative, we rather store $e[k] - m[k] \geq 0$. On the other hand, $m[k]$ is stored as $-m[k] \geq 0$. Since $s[k] \geq 2l[k] - m[k]$, we represent $s[k] - 2l[k] + m[k] \geq 0$ to induce smaller numbers. Many values in these arrays are expected to be small (and even smaller after these transformations), so we store them using a variable-length representation that uses fewer bits for smaller numbers. The representation we choose, called directly addressable codes (DACs) [Brisaboa et al. 2013], allows direct access to any cell value (we use the DAC variant that uses minimum space). Of course, the arrays pl and pr are stored using one bit per cell.

To further save space, only some nonterminals k will store this information. Let $R_k \rightarrow R_i R_j$. Then, it holds $m[k] = \min(m[i], e[i] + m[j])$, $e[k] = e[i] + e[j]$, $s[k] = s[i] + s[j]$, $l[k] = l[i] + l[j] + [1 \text{ if } pr[i] =' (' \ \wedge \ pl[j] =')']$, $pl[k] = pl[i]$, and $pr[k] = pr[j]$. These recurrences allow us computing the desired values for nonterminals R_k that do not store them. We use a technique [Navarro et al. 2011] that, given a parameter y, chooses a set of nonterminals that will store the array values, guaranteeing that we will never recursively expand more than y nonterminals in order to obtain any such value.

3.1.3. Storing the array C. Sequence $C[1, c]$ is stored as an array of nonterminals, that is, the corresponding preorder values in T_L, using $c \log r$ bits. In addition, the parentheses sequence P will be sampled every z positions. For the sth sample ($s \geq 0$), we store the following values.

(1) $C_p[s]$, the position in C whose expansion contains $P[zs + 1]$, that is, $C_p[s] = \min\{w, \sum_{v=1}^{w} |\mathit{str}(C[v])| > zs\}$.
(2) $C_o[s]$, the distance from $zs + 1$ to the beginning of $C[C_p[s]]$, that is, $C_o[s] = zs - \sum_{v=1}^{C_p[s]-1} |\mathit{str}(C[v])|$.
(3) $C_e[s] = \sum_{v=1}^{C_p[s]-1} e[C[v]]$, the cumulative excess up to the beginning of block $C[C_p[s]]$.
(4) $C_l[s]$, the number of leaves (occurrences of '10') up to the beginning of $C[C_p[s]]$, that is, $C_l[s] = l[C[1]] + \sum_{v=2}^{C_p[s]-1} (l[C[v]] + [1 \text{ if } pr[C[v-1]] =' (' \wedge pl[C[v]] =')'])$ if $C_p[s] > 1$, and $C_l[s] = 0$ otherwise.

On top of this array of samples, we form a balanced binary tree, where each node stores the minimum excess reached within the range of C it covers (the sth leaf covers the range $P[z(s-1)+1, zs]$). This excess is represented in absolute form, not relative to the range of blocks. Figure 5 illustrates the representation of C.

This sampled data adds $O((t \log t)/z)$ bits to the $c \log r$ bits used to store sequence C. If we choose $z = \Theta((t \log t)/c)$ and y large enough for the marked blocks be $O(r/\log t)$, then the space of our grammar representation is $(r + c) \log r + O(r + c)$, which is asymptotically equal to the space of a plain grammar-compressed representation the sequence $P[1, 2t]$. Since the c nonterminals in C expand to $2t$ characters, we will be able to scan the cells of C between two samples in $O(\log t)$ time on average[6]. We cannot bound the value of y required to have $O(r/\log t)$ samples, however, but our experiments will show that reasonable space/time tradeoffs are achieved.

[6]This can be made worst-case by regularly sampling $C[1, c]$ instead of $P[1, 2t]$, but this entails a binary search to find the sample corresponding to a position in P, and turns out to be slower than the sampling we chose, for the same space usage.

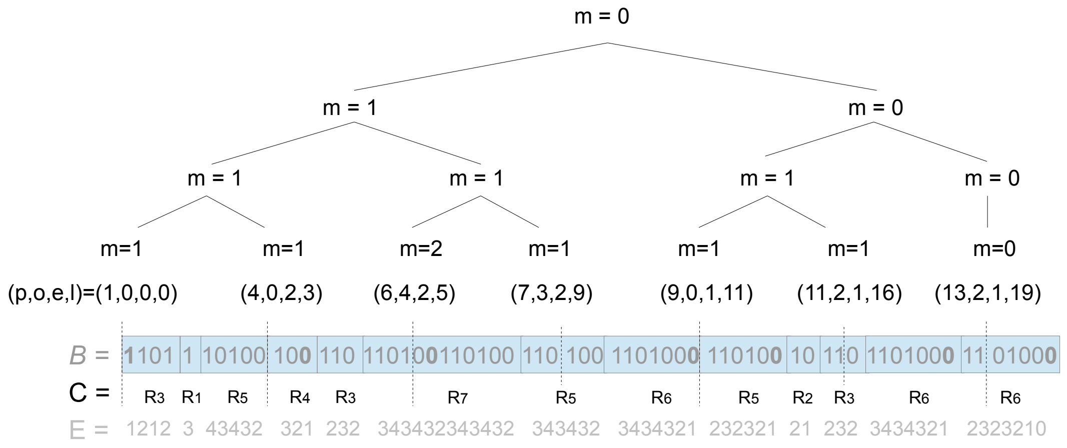

m = 0
m = 1
m = 0
m = 1
m = 1
m = 1
m = 0
m=1
m=1
m=2
m=1
m=1
m=1
m=0
(p,o,e,l)=(1,0,0,0)
(4,0,2,3)
(6,4,2,5)
(7,3,2,9)
(9,0,1,11)
(11,2,1,16)
(13,2,1,19)
B = 1101 1 10100 100 110 11010110100 110 100 1101000 110100 10 110 1101000 11 01000
C = R3 R1 R5 R4 R3 R7 R5 R6 R5 R2 R3 R6 R6
E = 1212 3 43432 321 232 343432343432 343432 3434321 232321 21 232 3434321 2323210

Fig. 5: Tree structure built on top of the C array of Figure 3, for a sampling step of $z = 10$. The positions of B in bold show where is the minimum excess reached, within each block.

3.2. Basic operations

We start by describing some basic operations on the GCT. The following procedure computes $E(p)$, the excess in $P[1,p]$, which is needed to compute $\textit{tDepth}(p) = E(p)$ and $\textit{preorder}(p) = (p + E(p))/2$.

(1) We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.
(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$, where we remind that $s[k]$ and $e[k]$ are the total length and excess, respectively, of $\textit{str}(R_k)$. We stop at the position $C[v]$ where q would exceed p if we processed v.
(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$ (i and j are found with the method described in Section 3.1.1). If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the excess e up to position $q = p$. Then we return $E(p) = e$.

If we store $e[]$ and $m[]$ values for all the nonterminals, then the sequential traversal in point 2 requires on average $O(\log t)$ time, as discussed at the end of Section 3.1.3. Point 3 takes time $O(\log^2 t)$, because the grammar is balanced and thus has height $O(\log t)$, and each time we expand $R_k \rightarrow R_i R_j$ in the downward traversal we take time $O(\log t)$ to find i and j with the representation of Section 3.1.1. Thus the total time is $O(\log^2 t)$. Instead, if we sample the values $e[]$ and $m[]$ with parameter y, then each computation of those values requires $O(y)$ symbol expansions, each of which still costs $O(\log t)$ time. This raises the total time to $O(y \log^2 t)$.

Another basic operation is to find the value of $P[p]$. This is necessary for *isLeaf* and *fChild*, and also participates in operations *nSibling* and *pSibling*. The recursion described for operation $E(p)$ ends up at a terminal in point 3, which is $P[p+1]$. Therefore, the following variation returns $P[p]$.

(1) We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$.

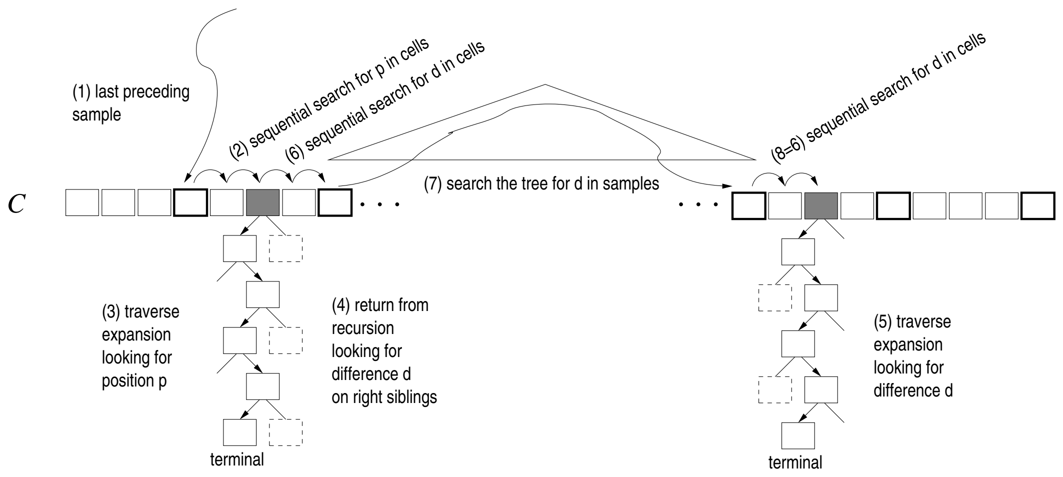

(1) last preceding sample
(2) sequential search for p in cells
(6) sequential search for d in cells
(7) search the tree for d in samples
(8=6) sequential search for d in cells
C
(3) traverse expansion looking for position p
(4) return from recursion looking for difference d on right siblings
(5) traverse expansion looking for difference d
terminal
terminal

Fig. 6: General scheme of the algorithm for *fwd*(p, d), assuming d is found after another sample. Sampled blocks have thick borders. Grayed blocks are nonterminals that become partially expanded; dashed ones are skipped after considering their $e[]$ and $m[]$ values. The middle tree is that of Figure 5.

(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$. We stop at the position $C[v]$ where q would reach p if we processed v.
(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] < p$, then we add $q \leftarrow q + s[i]$ and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, it is at position $q = p$ in P, so we return it.

3.3. Operations *fwd* and *bwd*

These are the two most important operations on the GCT, needed to implement *parent*, *nSibling*, *pSibling*, *ancestor*, *subtree*, and *tAncestor*. They also participate in operations *LCA*, *sLink*, *sAncestor*, *sDepth*, *sAncestor*, and *child*. We describe how to solve operation *fwd*(p, d), where p is a position in P. This follows the same spirit of the description of the operation on a balanced tree, recall Section 2.1. The scheme of the algorithm, with the corresponding steps, is depicted in Figure 6. Operation *bwd*(p, d) is analogous.

(1) We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.
(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$. We stop at the position $C[v]$ where q would exceed p if we processed v.
(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we have the excess e up to position $p = q$. Now we start looking for a negative difference d of excess to the right of position q.
(4) We traverse back (returning from recursion) the path in the grammar followed in point 3. If we went towards the right child R_j of a rule $R_k \rightarrow R_i R_j$, we just return. If, instead, we went towards R_i, then we check whether $d \leq -m[j]$. If so, the answer is within R_j, otherwise we add $q \leftarrow q + s[j]$ and $d \leftarrow d + e[j]$, and return to the parent in the recursion.

(5) If in the previous point we have established that the answer is within a nonterminal R_j, we traverse the expansion of $R_j \rightarrow R_l R_m$. If $d > -m[l]$, then we add $q \leftarrow q + s[l]$ and $d \leftarrow d + e[l]$, and continue recursively with R_m; else we continue recursively with R_l. When we reach a leaf, the answer is q.
(6) If in point 4 we return from the recursion up to the root symbol $C[v]$ without yet finding the desired excess difference d, we update $e \leftarrow e + e[C[v]]$ and scan the nonterminals $C[v+a]$ for $a = 1, 2, \ldots$, increasing $q \leftarrow q+s[C[v+a]]$, $e \leftarrow e+e[C[v+a]]$, and $d \leftarrow d + e[C[v + a]]$, until finding an a such that $d \leq -m[C[v + a]]$. At this point we look for the final answer within the nonterminal $C[v + a]$ just as in point 5.
(7) If we reach the next sampled position, $q \geq sz$, without yet finding the answer, we jump to the $\lfloor q/z \rfloor$th leaf of the balanced tree we built on the samples of C (the leftmost leaf is the 0th), and traverse it upwards until the current node has a right sibling whose (absolute) minimum value m satisfies $e - d \geq m$. Then we descend from that right sibling. If its left child's minimum value m_l satisfies $e - d \geq m_l$, we descend to the left child, otherwise to the right child.
(8) We eventually reach the s'th leaf of the tree, thus we know that the desired answer can be found from position $v \leftarrow C_p[s']$ in C. Note that $C[v]$ starts after position $q \leftarrow s'z - C_o[s']$ in P, and the excess up to q is $C_e[s']$. Thus we recompute $d \leftarrow d + C_e[s'] - e$ and $e \leftarrow C_e[s']$, and continue traversing the cells $C[v + a]$ sequentially, for $a \geq 0$, just as in point 6 (and eventually finishing as in point 5).

The complexity of this procedure is the same as for the simpler operations. The only new cost is $O(\log t)$ to traverse the balanced tree, which is not significant.

3.4. Operation *RMQ*

This operation is key for the important *LCA* and *sLink* suffix tree operations. To solve ***RMQ***(p, p') we traverse all the $O(\log t)$ grammar nodes between positions p and p' and locate the point where the minimum excess occurs.

(1) We compute $s \leftarrow \lfloor (p - 1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.
(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$. We stop at the position $C[v]$ where q would reach p if we processed v.
(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] < p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we are at position p and initialize $min \leftarrow e$. Now we update $e \leftarrow e+1$ if the terminal is a '(' or $e \leftarrow e - 1$ if the terminal is a ')'. Finally, we update $min \leftarrow \min(min, e)$ and $q \leftarrow q + 1$.
(4) We traverse back (returning from recursion) the path in the grammar followed in point 3. If we went towards the right child R_j of a rule $R_k \rightarrow R_i R_j$, we just return. If, instead, we went towards R_i, then we check whether $q + s[j] > p'$. If so, position p' is within R_j, otherwise we update $min \leftarrow \min(min, e + m[j])$, $q \leftarrow q + s[j]$, and $e \leftarrow e + e[j]$, and return to the parent in the recursion.
(5) If in the previous point we have established that p' is within a nonterminal R_j, we traverse the expansion of $R_j \rightarrow R_l R_m$. If $q + s[l] \leq p'$, then we update $min \leftarrow \min(min, e + m[l])$, $q \leftarrow q + s[l]$, and $e \leftarrow e + e[l]$, and continue recursively with R_m; else we continue recursively with R_l. When we reach a leaf, we have $q = p'$ and the minimum excess is min.
(6) If in point 4 we return from the recursion up to the root symbol $C[v]$ without yet reaching position p', we scan the nonterminals $C[v + a]$ for $a = 1, 2, \ldots$, updating $min \leftarrow \min(min, e + m[C[v + a]])$, $q \leftarrow q + s[C[v + a]]$, and $e \leftarrow e + e[C[v + a]]$, until

finding an a such that $q + s[C[v + a]] > p'$. At this point we complete the calculation of min within the nonterminal $C[v + a]$ just as in point 5.

(7) If we reach the next sampled position, $q \geq sz$, without yet reaching position p', we jump to the $\lfloor q/z \rfloor$th leaf of the balanced tree we built on the samples of C, and traverse it upwards until the the current node has a right sibling that covers position p'. Along the upward traversal, for each right sibling (that does not yet cover p') with minimum value m_r, we set $min \leftarrow \min(min, m_r)$. Once we find a right sibling that covers p' we descend from it. If its left child covers p', we just descend to the left, else we descend to the right and set $min \leftarrow \min(min, m_l)$, where m_l is the minimum value stored at the left child.

(8) We eventually reach the s'th leaf of the tree, thus we know that p' can be found from position $v \leftarrow C_p[s']$ in C. Note that $C[v]$ starts after position $q \leftarrow s'z - C_o[s']$ in P, and the excess up to q is $e \leftarrow C_e[s']$. Thus we continue traversing the cells $C[v + a]$ sequentially, for $a \geq 0$, just as in point 6.

(9) We finally have the min value, but not the position where it was reached. If min was set at a node of the balanced tree, we descend by its left or right children, whichever matches the minimum value of its parent, until reaching a leaf s''. Then we scan $k \leftarrow C[C_p[s''], \ldots]$, starting with $q \leftarrow s''z - C_o[s'']$, $e \leftarrow C_e[s'']$ and $m \leftarrow 0$, updating $q \leftarrow q + s[k]$, $m \leftarrow \min(m, e + m[k])$ and $e \leftarrow e + e[k]$, until we reach the value $e + m = \min$ for some k (before q reaches the next sampled block).

(10) Either because we computed it in point 9, or because min was reached within a nonterminal R_k starting after position q, we proceed as follows. If $R_k \rightarrow R_i R_j$, then if $m[k] = m[i]$, we continue recursively with R_i; otherwise we set $q \leftarrow q + s[i]$ and continue recursively with R_j. When we reach a terminal, the answer is $\boldsymbol{RMQ}(p, p') = q$.

3.5. Mapping with the CSA

The second group of operations of Table I requires the interaction with the CSA; see Sadakane [2007] for the details. From the GCT, what is missing is the ability to count the number of leaves up to some position $P[p]$, and to find the lth leaf in P. The storage of $l[k]$, $pl[k]$ and $pr[k]$ serves this purpose. We first show how to compute the number of leaves up to position p.

(1) We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the number of leaves up to that position is $l \leftarrow C_l[u]$. We set $pr \leftarrow pr[C[u-1]]$ (if $u > 0$, otherwise $pr \leftarrow')'$).

(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$, $l \leftarrow l + l[k] + [1 \text{ if } pr =' (' \wedge pl[k] =')']$, and $pr \leftarrow pr[k]$. We stop at the position $C[v]$ where q would exceed p if we processed v.

(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$, $l \leftarrow l + l[i] + [1 \text{ if } pr =' (' \wedge pl[i] =')']$, and $pr \leftarrow pr[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the number of leaves l up to position p. Then we return l.

Finding the lth leaf is the inverse of the above operation.

(1) We binary search C_l to find the largest s such that $C_l[s] < l$. This points to position $u \leftarrow C_s[s]$ in C, which starts after $q \leftarrow sz - C_o[s]$ in P. The number of leaves up to position q is $l' \leftarrow C_l[s]$, and we set $pr \leftarrow pr[C[u-1]]$ (if $u > 0$, otherwise $pr \leftarrow')'$).

(2) We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$, $l' \leftarrow l' + l[k] + [1 \text{ if } pr =' (' \wedge pl[k] =')']$, and $pr \leftarrow pr[k]$. We stop at the position $C[v]$ where l' would reach l if we processed v.

(3) Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \to R_i R_j$. If $l' + l[i] + [1 \text{ if } pr = '(' \wedge pl[i] = ')'] < l$, then we set l' to this value, update $q \leftarrow q + s[i]$, set $pr \leftarrow pr[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the number of leaves up to position q is $l' < l$ and that their number reaches l at position $q+1$. Then we return q, the starting position of the opening parenthesis that starts the lth leaf.

4. EXPERIMENTAL RESULTS AND DISCUSSION

We use various DNA collections from the Repetitive Corpus of *Pizza&Chili*[7]. On one hand, to study precisely the effect of repetitiveness in the performance of the suffix trees, we generate four synthetic collections of about 100MB: `DNA 1%`, `DNA 0.1%`, `DNA 0.01%`, and `DNA 0.001%`. Each `DNA` p`%` text is generated starting from 1MB of real DNA text, which is copied 100 times, and each copied base is changed to some other value with probability $p/100$. This simulates a genome database with different variability between the genomes.

On the other hand, we evaluate the performance of the suffix trees on a set of real repetitive DNA collections. This is important because repetitiveness may arise in other forms than just simple mutations on a copy of the sequence; for example it may involve block rearrangements. The least repetitive of our real collections is `Escherichia` (23 sequences adding up to 108MB, compressible by p7zip[8] to 4.72% of its original size), then `Para` (36 sequences adding up to 410MB, compressible by p7zip to 1.46%) is more repetitive, and `Influenza` (78,041 sequences adding up to 148MB, compressible by p7zip to 1.35%) is even more repetitive. We add a fourth collection that is the most repetitive, albeit it is not DNA but a versioned German Wikipedia article, `Einstein` (89MB, compressible by p7zip to 0.11%).

4.1. Space Usage

Figure 7 gives a space breakdown of our GCT representation for the 8 collections, in bps. The breakdown has five parts: (1) the RLCSA, which is built with parameters $blockSize = 32$ and $sample = 128$ to provide reasonable time performance; (2) the LCP representation; (3) the representation of the rules R of the GCT; (4) the representation of sequence C of the GCT; (5) the extra data we store for R and C associated to samples. For this last part, we tested various values of $y \in \{2^0, 2^1, 2^2, 2^4, 2^8\}$ and $z \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$. Obviously, this is the only part of the space that changes with y and z. We used the balanced version of RePair, which consistently gave us better results.

In the synthetic DNA collections, the space decreases as repetitiveness increases. The fixed part of the structures (without the sampling data on R and C) goes from about 0.85 bps on the most repetitive collection to about 4.7 bps when the mutation rate reaches 1%. Note that, from this space, about 0.6 bps from the RLCSA are fixed and insensitive to repetitiveness; this is the space used by the RLCSA samples. Components *LCP*, R and C decrease monotonically with repetitiveness.

The space for the R and C samplings varies significantly with parameter z, but not so much for y. This suggests that the rule sampling does not decrease the space significantly, whereas it does increase the time. Our best space/time combinations generally store the rule data for all the nonterminals, and use z to obtain space/time tradeoffs.

On the real data, the situation is more or less the same. Using reasonable values for z, the space is about 7 bps for `Escherichia`, the least repetitive collection. However,

[7]`http://pizzachili.dcc.uchile.cl/repcorpus`
[8]An LZ77-like compressor with a huge buffer, thus it captures most of the actual repetitiveness of the sequence, see `http://www.7-zip.org`.

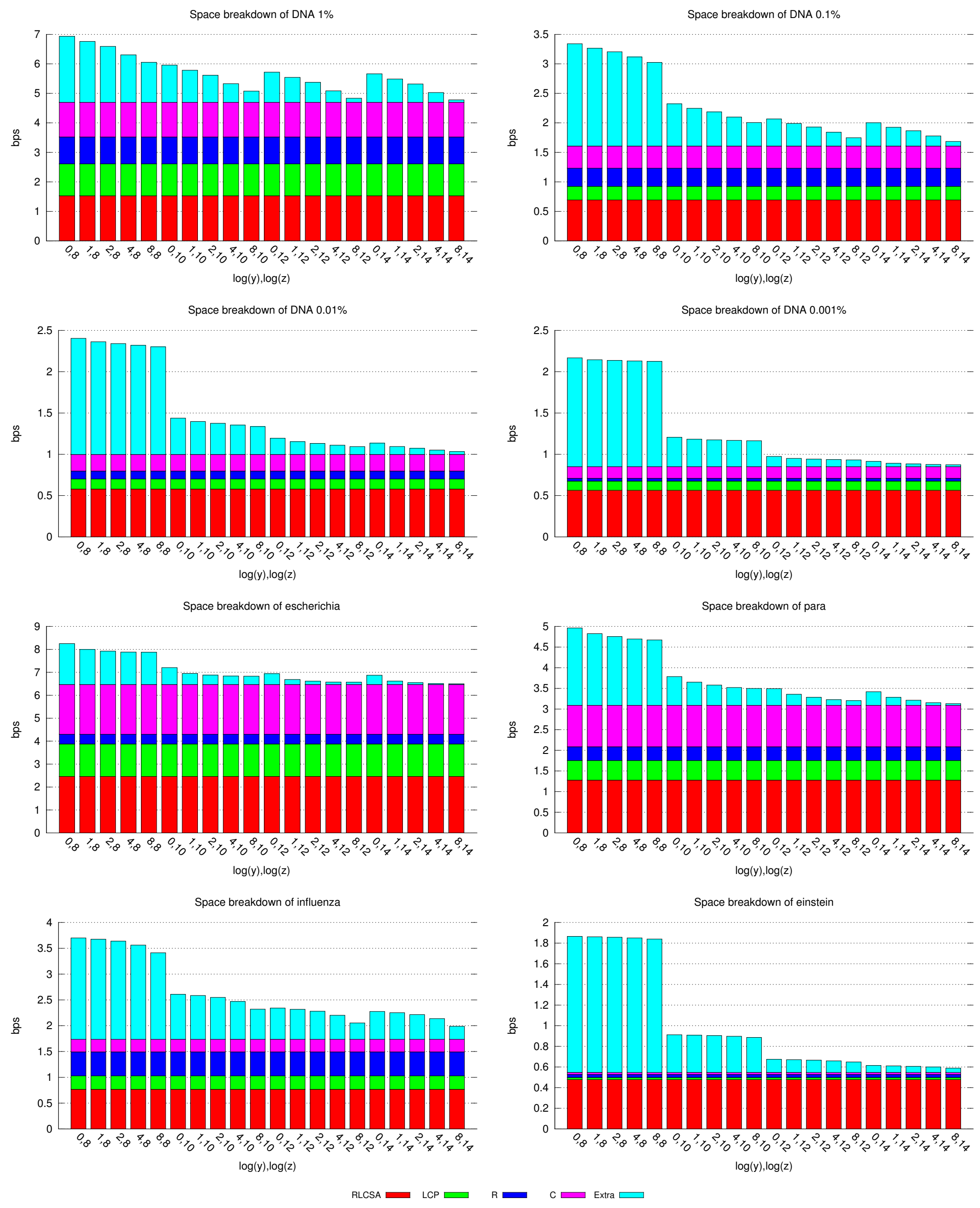


Fig. 7: Space breakdown of our GCT representation for the different collections and combinations of parameters y (rule sampling) and z (sampling of C).

it decreases to about 3.3 bps on `Para` and to 2.3 on `Influenza`, which is much more repetitive. On `Einstein`, the most repetitive collection, this space is below 0.7 bps.

4.2. Space-Time Performance of Operations

We compare space and time performance of our GCT with previous CSTs, for a number of suffix tree operations. The CSTs considered are the following.

GCT. Our new suffix tree representation. We used various combinations of parameters y and z, obtained a cloud of points, and chose the dominant ones. In most cases, this implies leaving y at sampling every nonterminal and using z to reduce the space. The RLCSA parameters are fixed to 32 for the sampling of Ψ and 128 for the text sampling (the one that affects the computation of suffix array entries).
Sada. Sadakane's CST [Sadakane 2007] adapted to repetitiveness but without including our new grammar-compressed tree topology (the actual index is too large to be of interest in this comparison). That is, we use the RLCSA as the suffix array, the compressed-bitvector H for the LCP, and the plain representation that uses 2 bits per node [Navarro and Sadakane 2014; Arroyuelo et al. 2010] for the topology. This allows us to measure the effect of our grammar-compressed topology in time and space. We use sampling 32 for Ψ and 64, 128, and 256 for the text sampling.
SCT3. The fastest CST for general collections among those that use reasonable space [Ohlebusch et al. 2010]. It is also the most compact CST implemented in the SDSL library [Gog 2015] (called `cst_sct3` in SDSL). For the CSA it uses an FM-index on Huffman-shaped wavelet trees [Ferragina et al. 2007], which makes it small and fast on DNA. It uses a non-compressed bitvector H to represent the LCP, and a structure of $3n$ bits to solve *PSV*/*NSV* operations. The tree topology is not represented. The bitvector samplings is set to 63, the sampling to extract text to 63, and the text sampling to 32, 64, and 128. For the rest, it was compiled with the default configuration of SDSL.
NPR-Repet. The only previous CST designed for repetitive collections [Abeliuk and Navarro 2012; Abeliuk et al. 2013]. We choose the best point between using balanced or unbalanced RePair in each case. They run over the RLCSA, with sampling 32 for Ψ and 64, 128, 256, and 512 for the text.
NPR. The smallest CST for general collections that achieves times within microseconds [Cánovas and Navarro 2010; Abeliuk et al. 2013]. Among their many variants, we use the so-called FMN-RRR, which uses the least space. To make it more space-competitive in this scenario, we change its suffix array to the RLCSA, with the same sampling choices of NPR-Repet.
FCST. The smallest CST for general collections [Russo et al. 2011]. The FCST is much slower than NPR; its times are in the range of the milliseconds, close to those of NPR-Repet. The FCST also uses an FM-index on wavelet trees [Ferragina et al. 2007] as its suffix array.

We exclude the faster and larger variants of NPR [Cánovas and Navarro 2010; Abeliuk et al. 2013], as they represent LCP values directly and these become very large on repetitive collections (≈ 27 bps only the LCPs!). Other larger variants implemented in SDSL are also disregarded in this comparison.

We note that not all the previous CSTs implement all the operations, so they may not appear in some plots. In addition, we were unable to build NPR-Repet on the most repetitive dataset, `DNA 0.001%`, because its grammar-compression algorithm on the differential *LCP* array crashed.

We ran the experiments in an isolated Intel(R) Core(TM) i7-3820 running at 3.60GHz with 62GB of RAM memory. The operating system is GNU/Linux, Ubuntu 12.04, with kernel 3.2.0-68-generic.x86_64. All our implementations use a single thread and all of them but FCST are coded in `C++` (FCST is in `C`). The compiler is `gcc` version $4.6.3$, with `-O9` optimization flag set (except SCT3, which uses its own set of optimization flags).

We averaged each data point over 10,000 random queries, following for each distinct operation the methodology described in previous work [Navarro and Sadakane 2014; Abeliuk et al. 2013] to choose the suffix tree nodes on which operations are carried out.

4.2.1. Space. Let us use Figure 8 to discuss the space usage of the indexes. The general-purpose indexes are mostly insensitive to repetitiveness (except because in some of those we used the RLCSA as the suffix array). Even with the sparsest samplings used, Sada takes up to 10 bps on the least repetitive collections and then decreases to 7 bps on the most repetitive ones. SCT3 (which does not use the RLCSA) always uses about 5.5–7 bps. NPR (which uses an RLCSA) goes from 6.5 bps on the least repetitive collections and to 4.5 bps on the most repetitive ones. Finally, the FCST (which also does not use an RLCSA) is the only one that increases space with repetitiveness, rarely exceeding 4 bps but reaching 5.3 bps on `Einstein`. The reason is that repetitive collections induce deeper suffix trees. Since the FCST samples nodes at regular intervals across *sLink* paths, a deeper suffix tree entails longer paths and thus more samples (up to some maximum guaranteed limit).

The repetitiveness-oriented CSTs use significantly less space, NPR-Repet being always smaller than GCT. The GCT becomes, in broad terms, more competitive with NPR-Repet as repetitiveness increases. While, on the least repetitive `DNA 1%`, NPR-Repet can use as little as 2.8 bps, which is about 60% of the GCT space, the ratio raises to 80% already for `DNA 0.01%`. On the real texts, instead, the ratio stays around 60%, but for the most repetitive `Einstein` both indexes use basically the same space.

Note that, on the least repetitive collections, the repetitiveness-oriented CSTs are not interesting anymore: On `DNA 1%` and `Escherischia`, the FCST is already smaller than the GCT (albeit much slower), and uses about the same space and time of NPR-Repet.

The comparison between GCT and Sada shows that compressing the parentheses reduces the space by 2–6 bps, the impact being larger on the more repetitive collections. On those, this difference dominates the total space of the structures, for example Sada is about 7 times larger than the GCT on `DNA 0.001%` and on `Einstein`.

The impact on the times is analyzed next. For the GCT, we will comment about the choice of parameter that reaches its "sweet point", which is roughly the left-to-right point where the time ceases to decrease abruptly and stabilizes. This is still a choice of good space usage.

4.2.2. Direct tree operations. Figures 8 to 12 show the time-space performance for operations *fChild* (requiring just an access to the parentheses), *tDepth* (requiring simple parenthesis operations), *nSibling*, *parent* and *tAncestor* (requiring the more complex *fwd* and *bwd* operations on the parentheses). For *tAncestor* we test with a random depth between 1 and the tree node depth.

Direct tree operations are particularly fast when the topology is represented with parentheses. This is the case of Sada and the GCT. In the first case the operation times goes from one nanosecond (ns) to at most one microsecond (μs). The faster ones, running in at most 10 ns, are *fChild*, *tDepth*, and *parent*. Instead, *nSibling* and *tAncestor* are slower, requiring 0.5–1 μs.

The GCT is not so fast because it compresses the topology, but still it performs well. It solves *fChild*, *tDepth* and *parent* in 5–10 μs, *tAncestor* in 10–30 μs, and *nSibling* 20–50 μs. That is 1–3 orders of magnitude slower than a plain parentheses representation.

Instead, the operations are 2–3 orders of magnitude slower on NPR, which uses much more space than GCT but does not store the tree topology. NPR requires 100–700 μs for operations *fChild*, *nSibling*, and *parent*, except on `Influenza` and `Einstein`, where for unclear reasons the times drop to 10–80 μs.

Lacking an explicit topology, *tDepth* and *tAncestor* can only be solved via successive *parent* operations until reaching the root or the desired depth difference. This makes these two operations way slower on the other indexes. For *tDepth* the time of NPR reaches 0.7–5 *ms*, and for *tAncestor* it reaches 2–50 *ms* (and 50–300 μs on the two collections where it is faster).

If we consider NPR-Repet, which does not store the tree topology and in addition is optimized for repetitiveness (reaching less space than the GCT), the times jump one or two orders of magnitude further: *fChild*, *nSibling*, and *parent* require 0.6–10 *ms* (0.3 *ms* on `Einstein`), *tDepth* takes 50–300 *ms* (10 *ms* on `Einstein`), and *tAncestor* uses 7–500 *ms*. Therefore, the only previous index that is smaller than the GCT on repetitive collections is 2–4 orders of magnitude slower than it. The choice of including the parentheses, even if highly compressed and slow to use, definitely pays off.

SCT3 does not represent the topology, but uses $3n$ bits to speed up the operations *PSV*/*NSV* on the LCP values. As a consequence, it uses more space than NPR, but it performs significantly faster. For *fChild*, *nSibling* and *parent*, it takes 0.2–2 μs, which is 1–2 orders of magnitude faster than GCT (but still way slower than Sada, which uses plain parentheses). However, it is also 1–2 orders of magnitude slower than GCT for *tDepth* and *tAncestor*, where it takes 10–40 μs and 0.2–2 *ms*, respectively (for unclear reasons, SCT3 is much slower on `Para`).

Finally, the FCST takes 0.6–7 μs on operations *fChild*, *nSibling* and *parent*, and 2–50 *ms* on *tDepth* and *tAncestor*. This is also several orders of magnitude slower than the GCT.

4.2.3. Operation LCA. This is the most complex among the operations that only need the topology of the suffix tree. Figure 13 shows that the GCT uses 30–100 μs to solve it. NPR requires 0.3–1 *ms* in most cases, and 30–200 μs on `Influenza` and `Einstein`. On the other hand, NPR-Repet requires 0.7–10 *ms*.

The heaviest part of this operation is a *RMQ*. Both SCT3 and Sada have explicit structures to carry out this operation, thus they solve it fast, in 4–5 μs. The FCST is also particularly fast on this operation: 5–20 μs. The reason is that *LCA* is a core operation for the FCST, so it is solved most efficiently and is the base for the other operations.

4.2.4. Operation sLink. The suffix link operation is the first we study that is specific of suffix trees, and can be considered as the one that distinguishes suffix trees from other digital trees. Operation *sLink* requires several tree operations and interacting with the RLCSA. For the GCT and Sada, it requires mapping the node to its suffix array interval (which involves *fwd* and counting leaf nodes up to a position in P), then computing the native function Ψ of the RLCSA [Mäkinen et al. 2010] (or the inverse of LF in the FM-index [Ferragina et al. 2007]) for both extremes of the interval, then maping them back to suffix tree leaves (which requires finding the lth leaf in the tree), and finally computing an *LCA* operation. The GCT requires 60–300 μs for operation *sLink*, whereas Sada needs only 2–5 μs, profiting from its faster tree operations.

On the structures that do not use explicit tree topologies, the node identifier is directly the suffix array interval, and thus all what is needed is to compute Ψ on both extremes of the interval and then an *LCA* operation on the resulting positions. In the case of NPR-Repet and NPR, the time for *LCA* dominates the others: 0.6–40 *ms* and 0.2–1 *ms* (40–100 μs on `Influenza` and `Einstein`), respectively. In the case of the FCST, operation *LCA* is fast but the other operations are not so much, driving the time to 0.2–1.5 *ms*. SCT3 takes 2–5 μs, being only slower than Sada.

4.2.5. Operation sDepth. This operation computes the string depth of a node, and is crucial for other suffix tree operations. In the GCT and Sada it requires mapping the

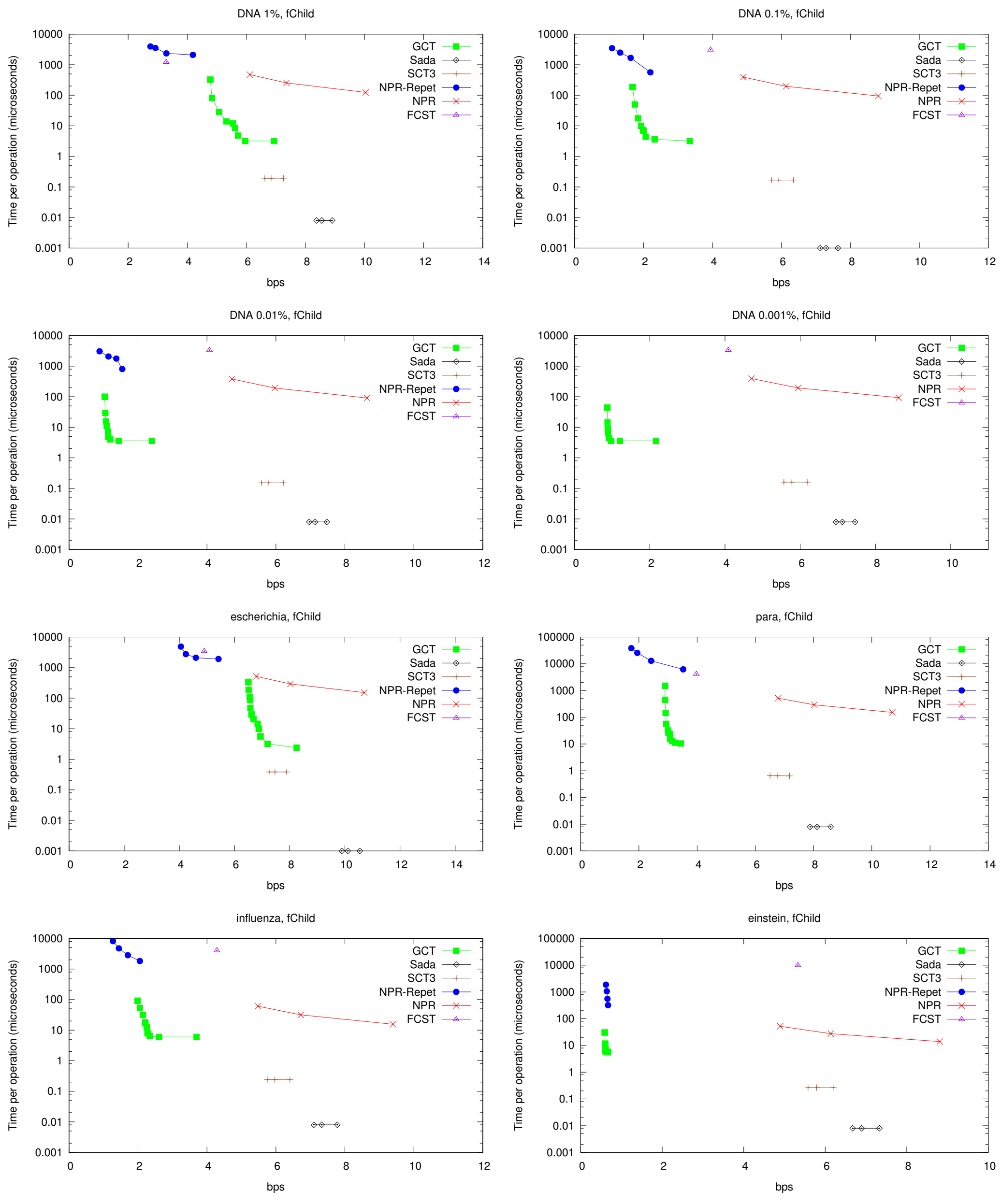


Fig. 8: Space-time tradeoffs for operation *fChild*.

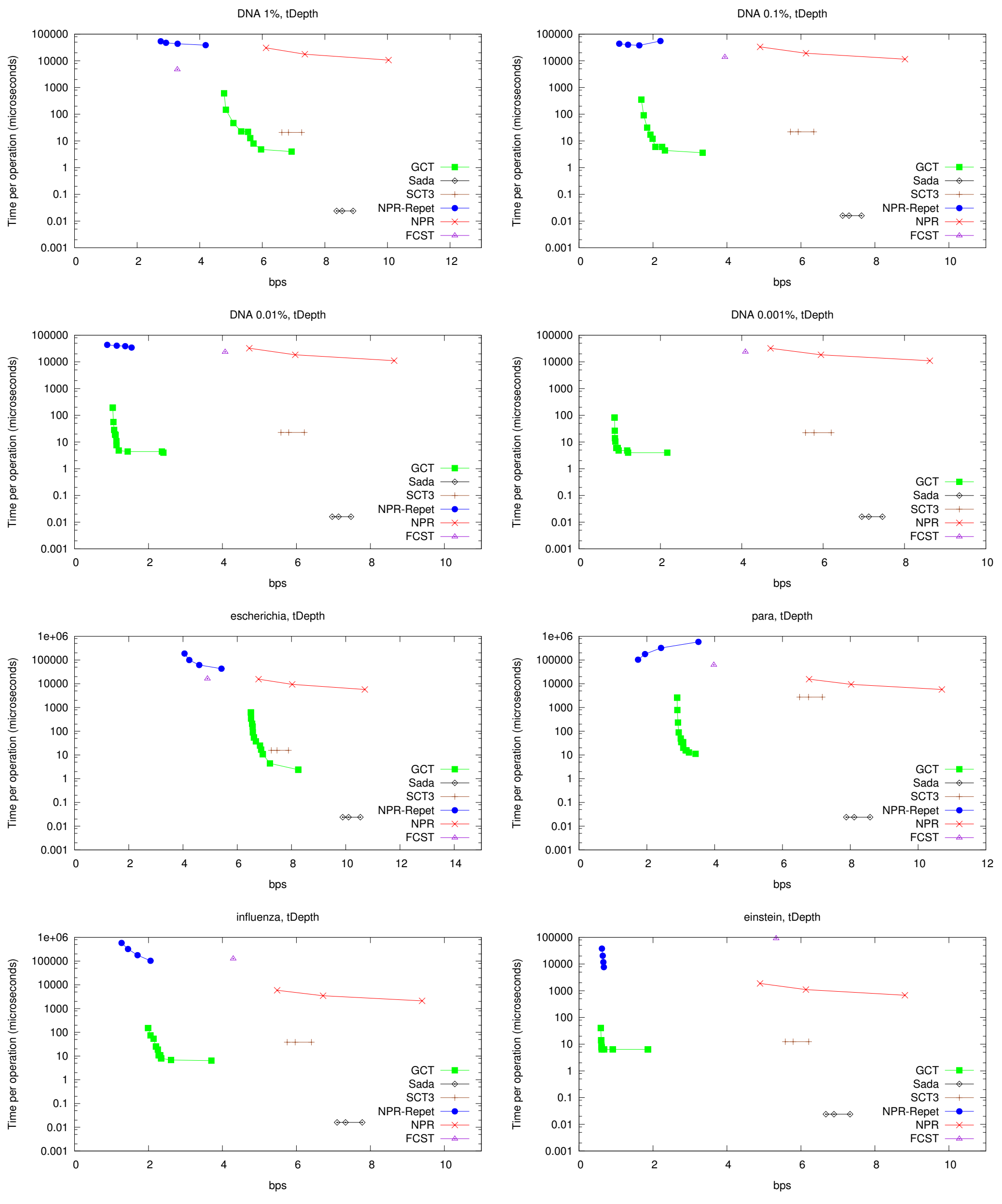


Fig. 9: Space-time tradeoffs for operation *tDepth*.

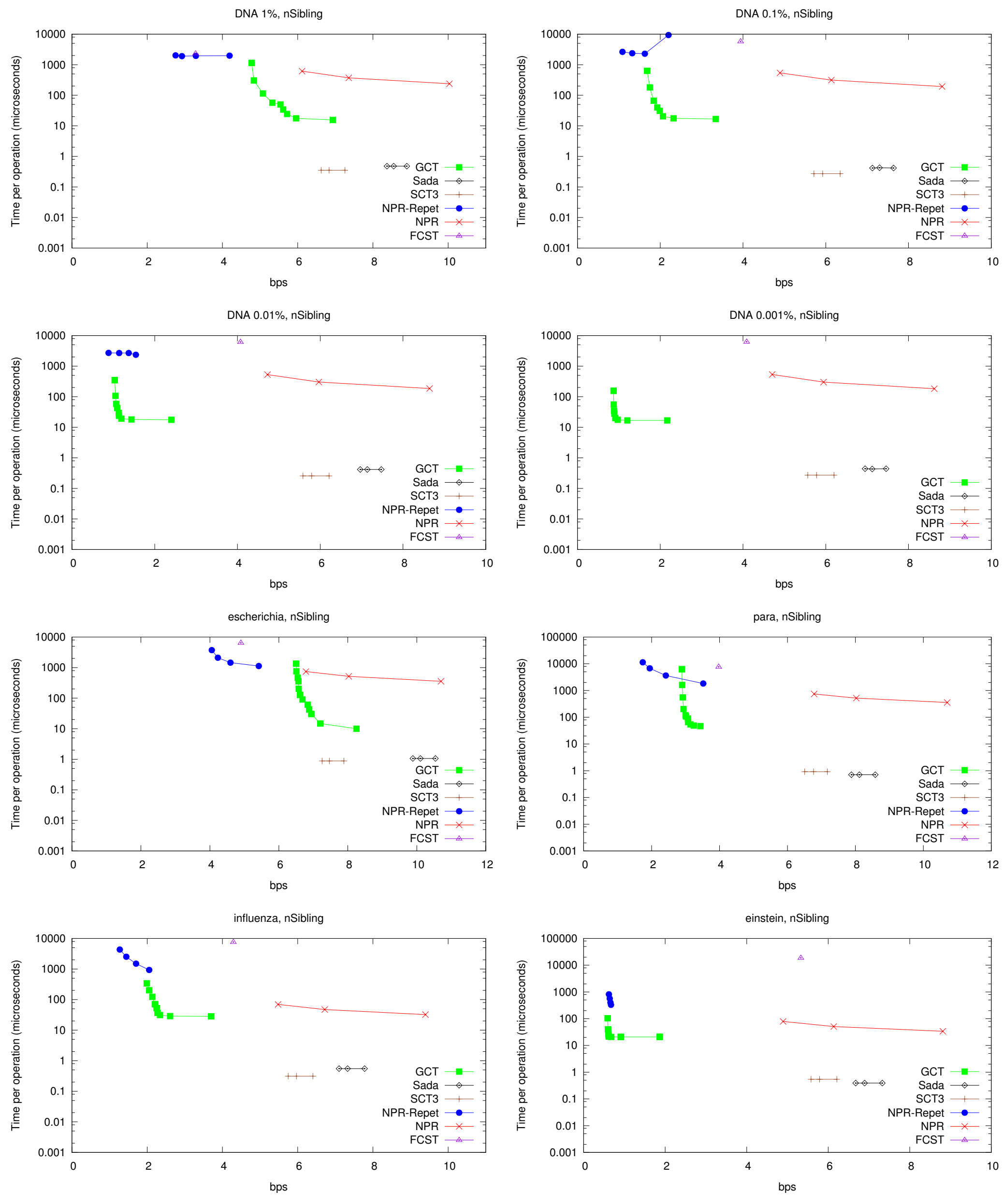


Fig. 10: Space-time tradeoffs for operation *nSibling*.

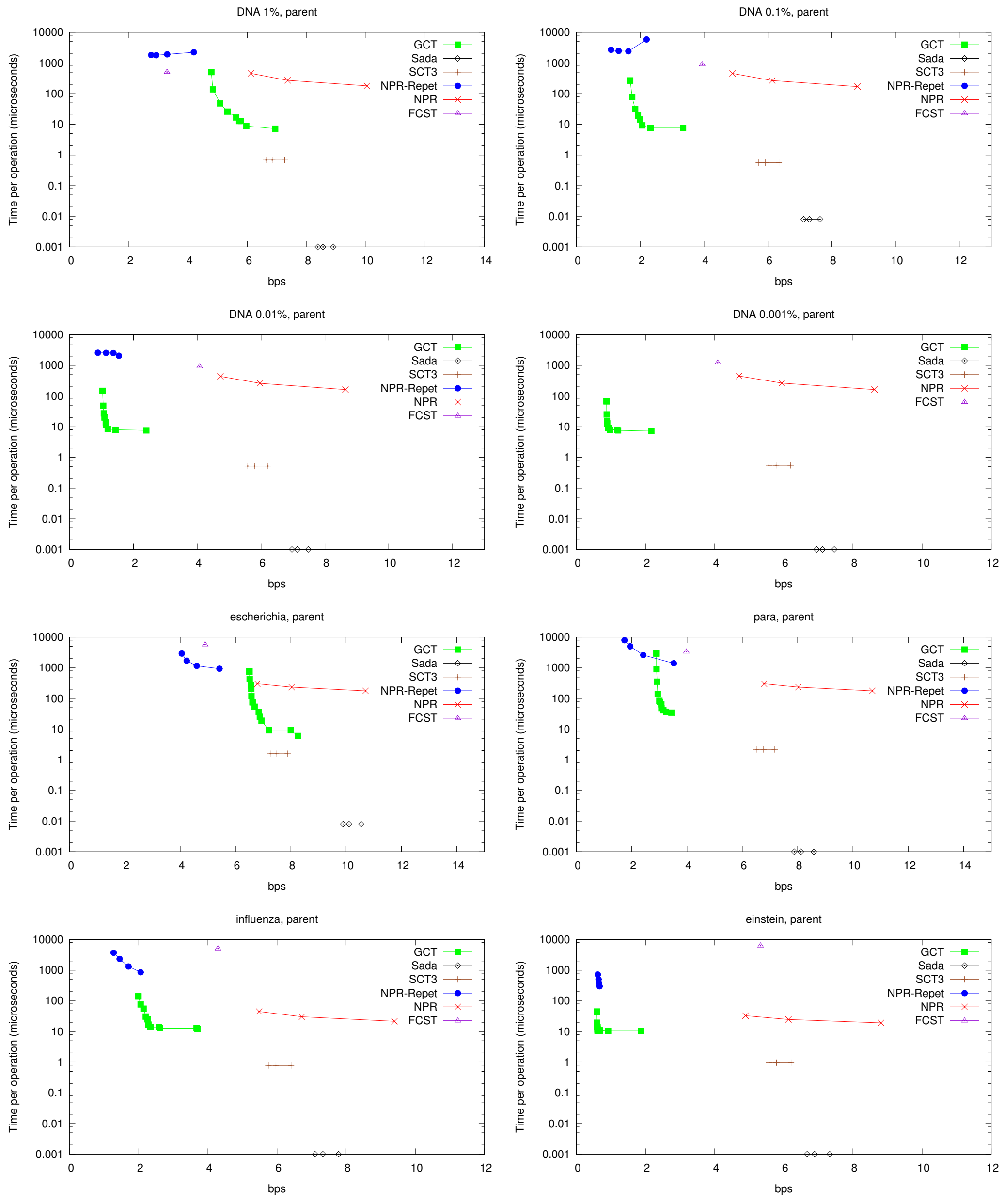


Fig. 11: Space-time tradeoffs for operation *parent*.

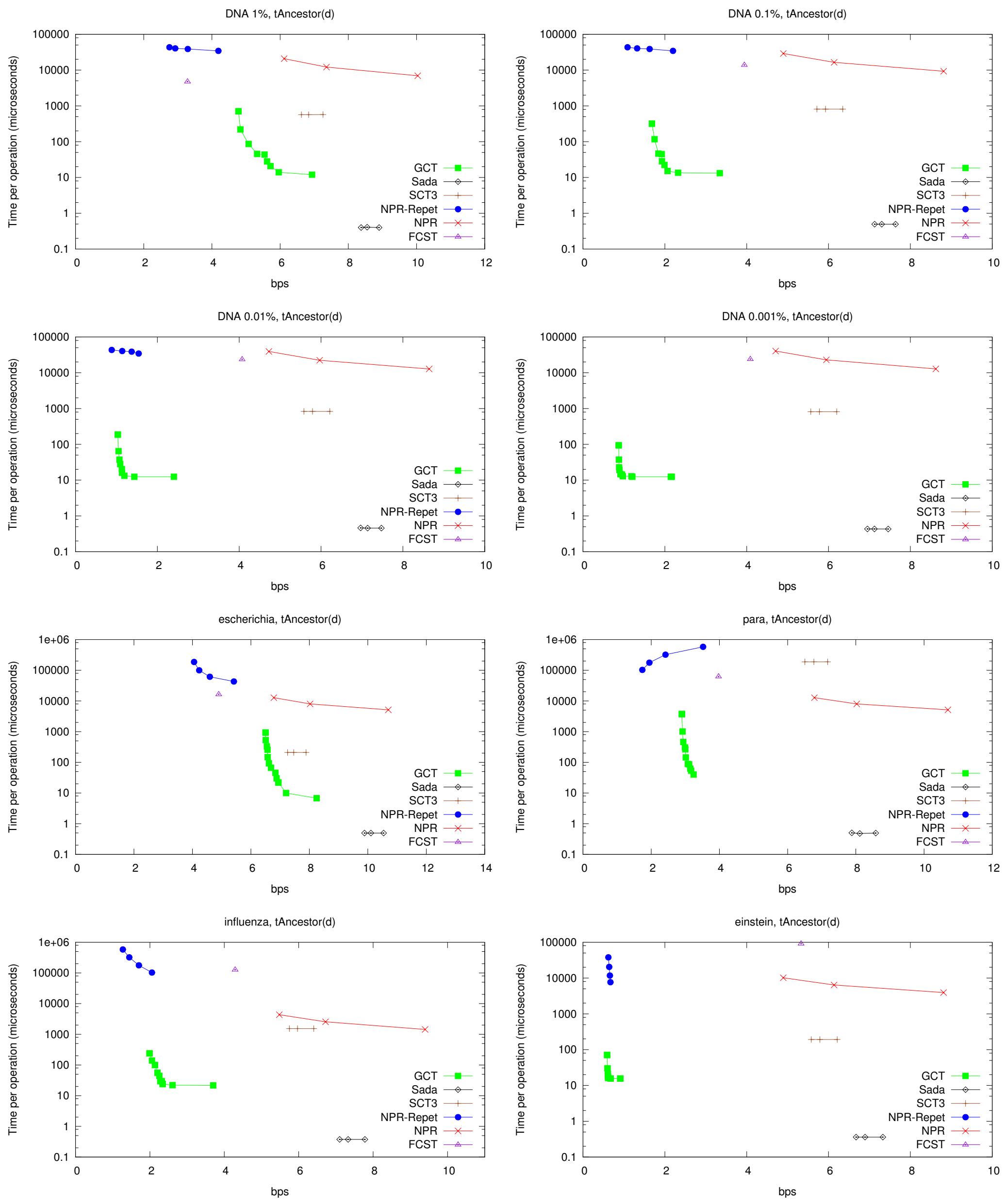


Fig. 12: Space-time tradeoffs for operation *tAncestor*.

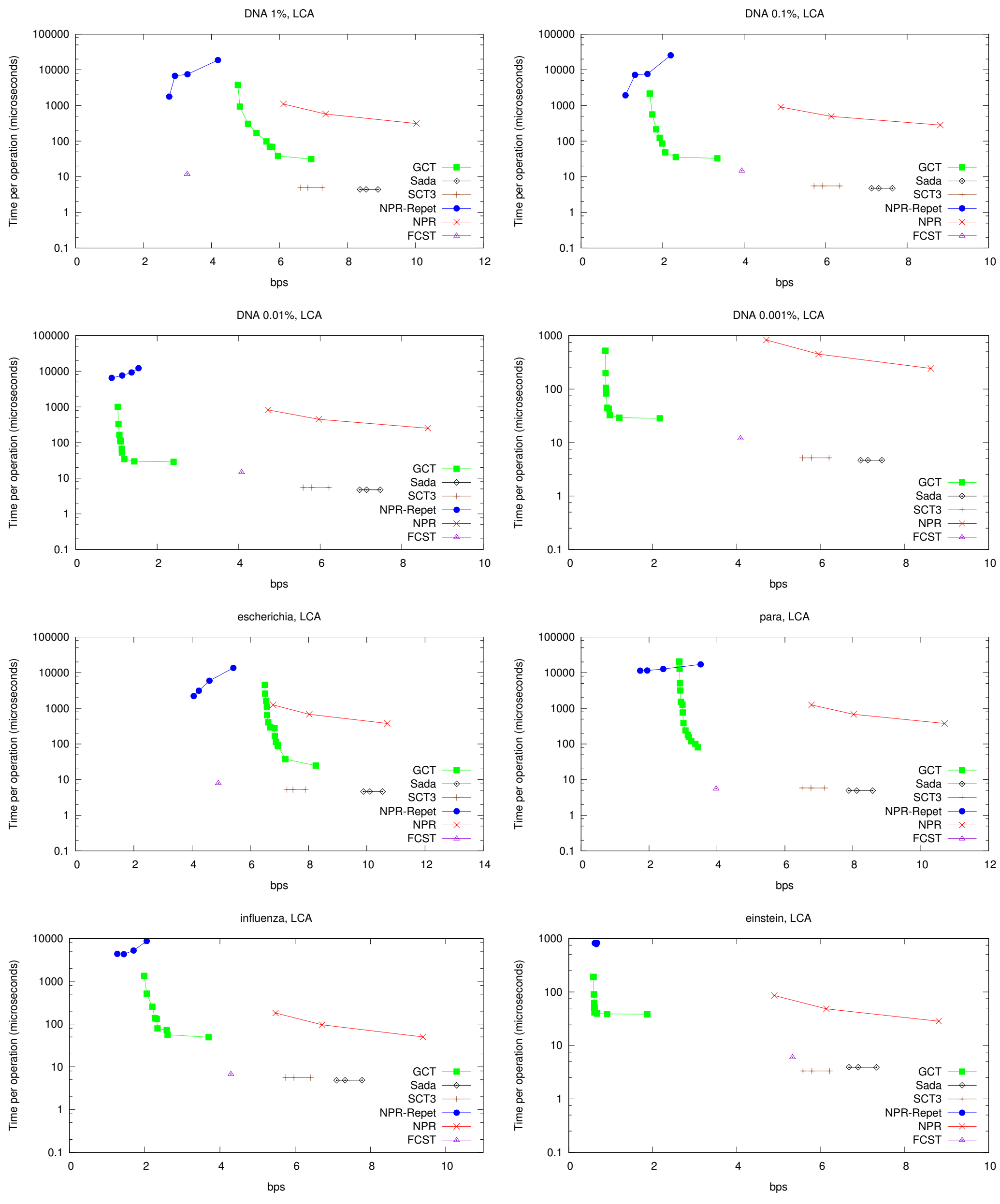


Fig. 13: Space-time tradeoffs for operation *LCA*.

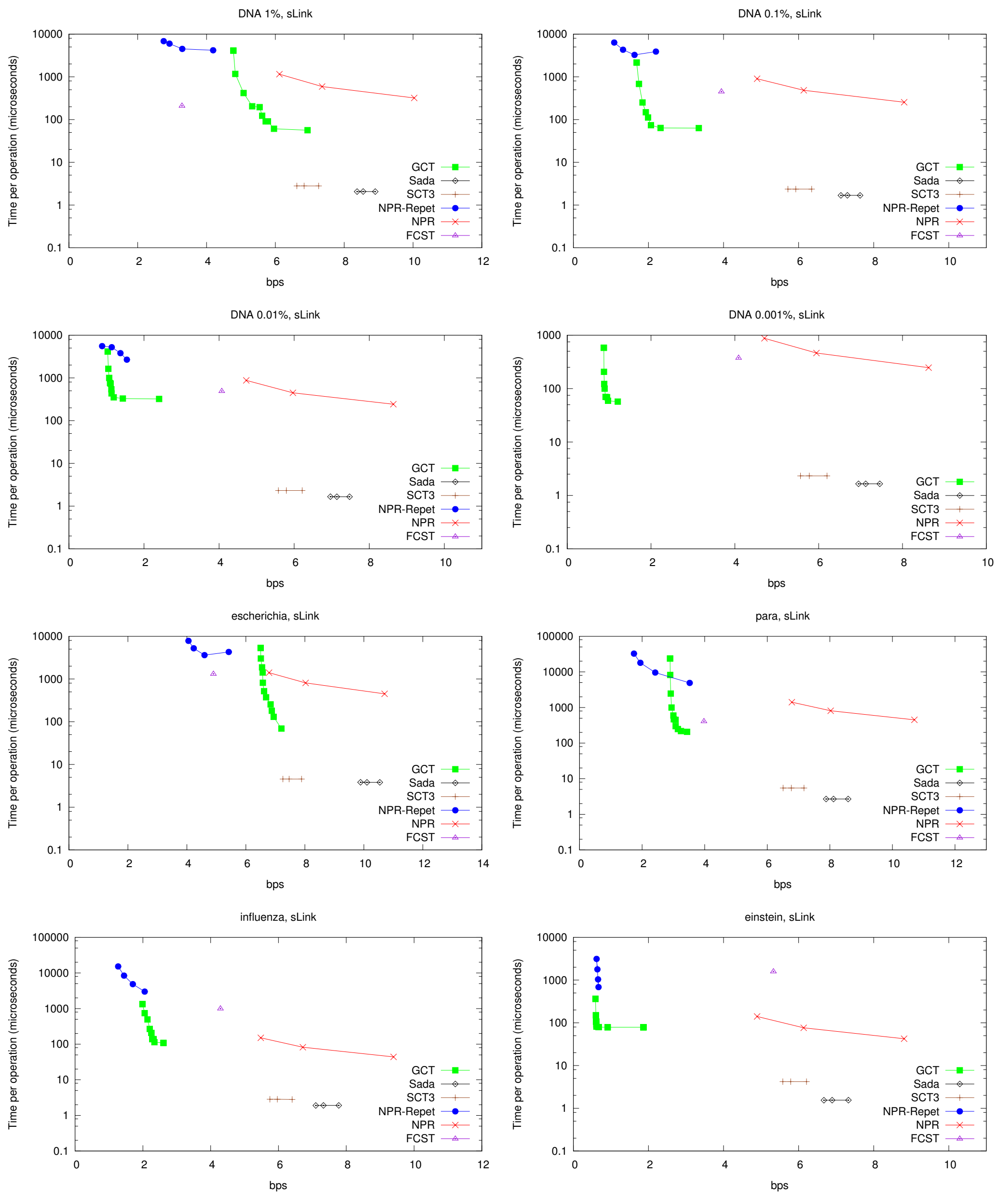


Fig. 14: Space-time tradeoffs for operation *sLink*.

second child of the node to the CSA (and thus it involves the corresponding *fwd* and leaf counting operation), whereas in NPR, NPR-Repet and SCT3 it requires a *RMQ* operation. Then, both kinds of structures must accesses the *LCP* data, which implies using the bitvector H plus the locating functionality of the RLCSA or FM-index. Therefore, this is the first operation where the text sampling of the suffix array plays a role in the time performance, actually dominating the overall time in various cases. The FCST, instead, implements *sDepth* as a core operation.

The operation takes 50–100 μs on the GCT, 60–1000 μs on NPR-Repet, and 60–100 μs on NPR (6–10 μs on `Influenza` and `Einstein`). Sada and SCT3 require 10–100 μs, the tradeoff being also dominated by the text sampling. The FCST takes 0.7–3 *ms*.

4.2.6. Operation sAncestor. This finds the ancestor of the node with the given string depth (we test with a depth chosen at random between 1 and the node string depth). On GCT and Sada, which can compute *tAncestor* fast, this operation can be carried out via a binary search on *sDepth* using *tAncestor*. Thus it is computed in 250–700 μs on the GCT, and in 50–300 μs on Sada.

On the SCT3 and FCST, the operation must be computed via successive *parent* operation and measuring *sDepth* at each node. Therefore, it is more expensive: 0.3–3 *ms* on SCT3 and 75–300 *ms* on FCST.

Instead, this operation is almost native on NPR and NPR-Repet [Abeliuk et al. 2013], since they use on the LCP array a structure similar to the one we use on the excess of the parentheses. However, it still needs to compute some *sDepth* values on unsampled blocks of the LCP array, and this cost dominates. NPR takes 250–1000 μs (except 40–200 on `Influenza` and `Einstein`) and NPR-Repet takes 1–10 *ms*.

4.2.7. Operation letter. This is a simple operation exclusive of suffix trees. It gives the ith letter of the string represented by a node (we test $i = 4$). On the GCT and Sada, it requires mapping to the suffix array and computing Ψ^{i-1} on the RLCSA or LF^{-i} on the FM-index (this is usually faster than computing a suffix array cell). The GCT solves it in 4–10 μs and Sada in 0.75–1 μs.

The other CSTs use direct suffix array ranges, and thus do not need the mapping step. As a result, their time depends only on the CSA they use, and are faster than the GCT, and even than Sada: NPR-Repet uses 2–4 μs, NPR uses 0.3–1 μs, and FCST takes 4 μs. The operation is not implemented in SCT3, but as it depends on the FM-index used, it should be close to 1–4 μs as well.

4.2.8. Operation child. Finally, the most complex operation is *child*, which descends to a child by an edge labeled with a given letter. It must first compute *sDepth* and then traverse linearly the children of the node, computing *letter* for each until finding the desired one.

The operation takes 0.3–1 *ms* on the GCT, 1–10 *ms* on NPR (but 0.3–1 *ms* on `Influenza` and `Einstein`), 2.5–6 *ms* on FCST, 2–30 *ms* on NPR-Repet, 70–1000 μs on Sada, and 30–200 μson SCT3 (with the exception of `DNA 0.001%`, where it reaches almost 3 *ms*). Only SCT3, which has a fast implementation of *NSV* to find the successive children, and Sada, are faster than GCT.

4.2.9. Other operations. We have left out other less important operations from the experiments: *root* is trivial in all implementations; *preorder* is similar to *tDepth* for the GCT, and not possible to implement in the other schemes, which do not maintain the tree topology; *pSibling* is similar to *nSibling*; *isLeaf* costs the same as *fChild* on the GCT and is instantaneous on the others (as they use suffix array intervals as suffix tree node identifiers, and thus leaves correspond to intervals of length 1); *ancestor* is similar to *nSibling* and is instantaneous on the others (as it involves checking containment of intervals); *subtree* is also similar to *nSibling* and cannot be implemented

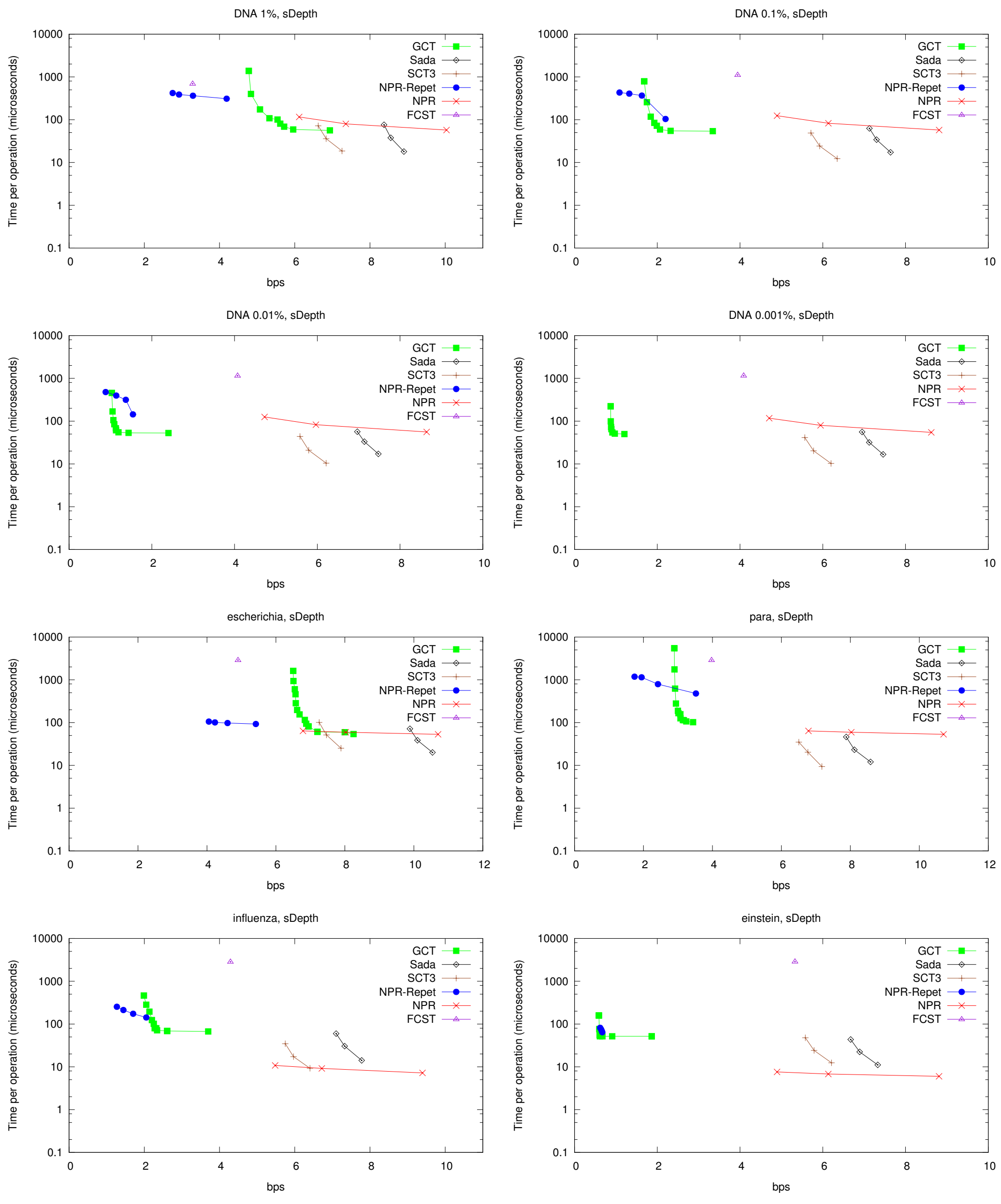


Fig. 15: Space-time tradeoffs for operation *sDepth*.

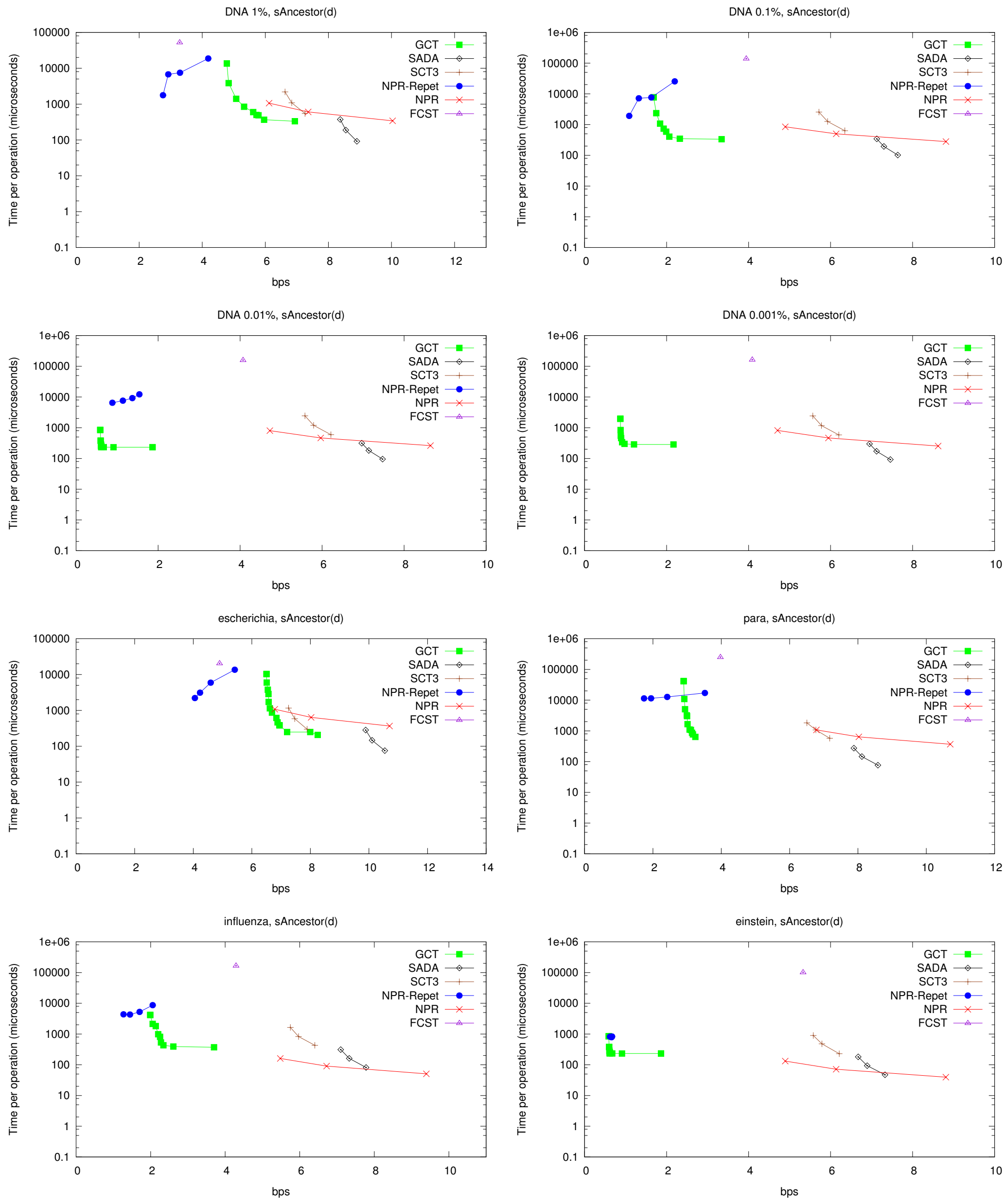


Fig. 16: Space-time tradeoffs for operation *sAncestor*.

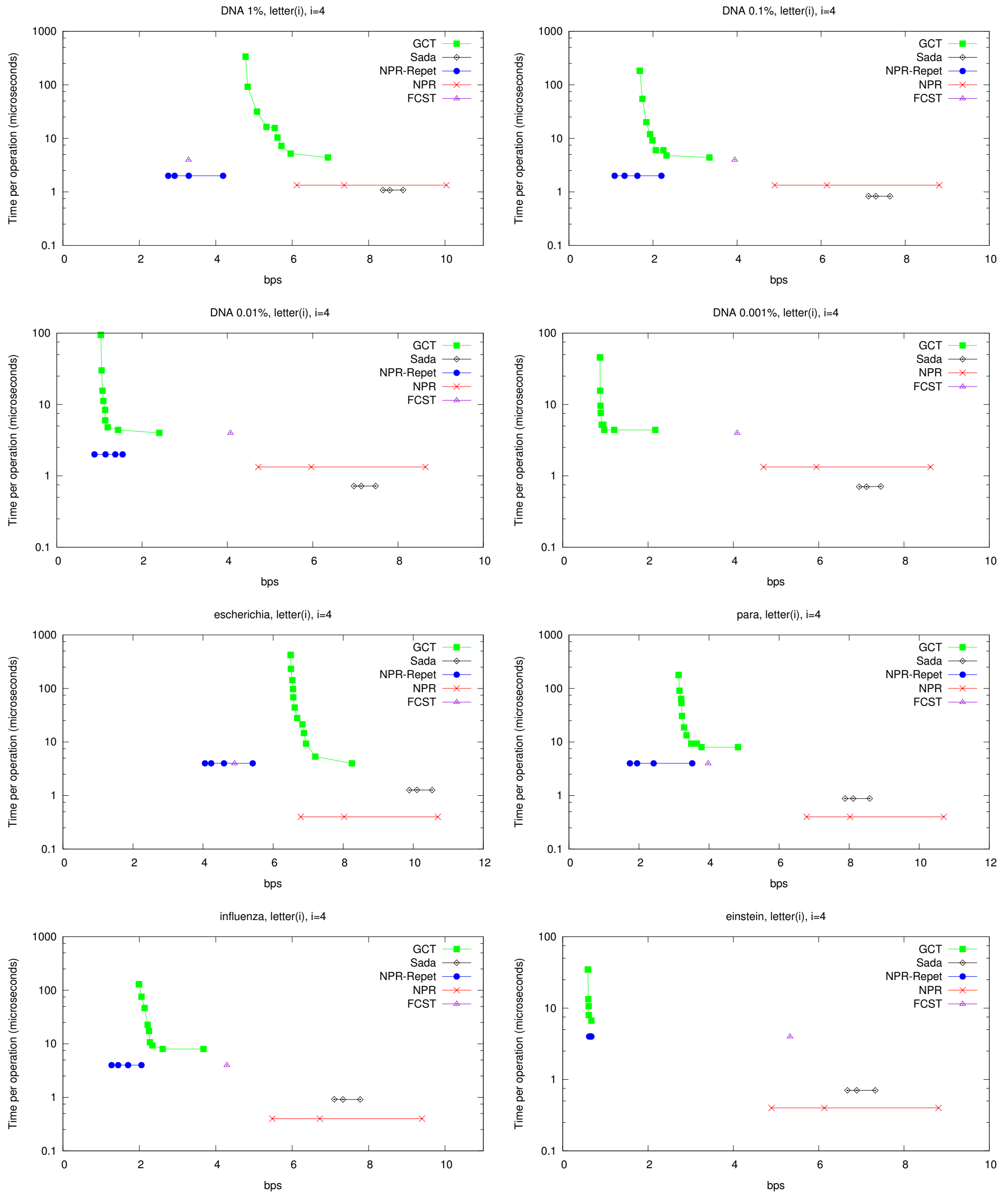


Fig. 17: Space-time tradeoffs for operation *letter*.

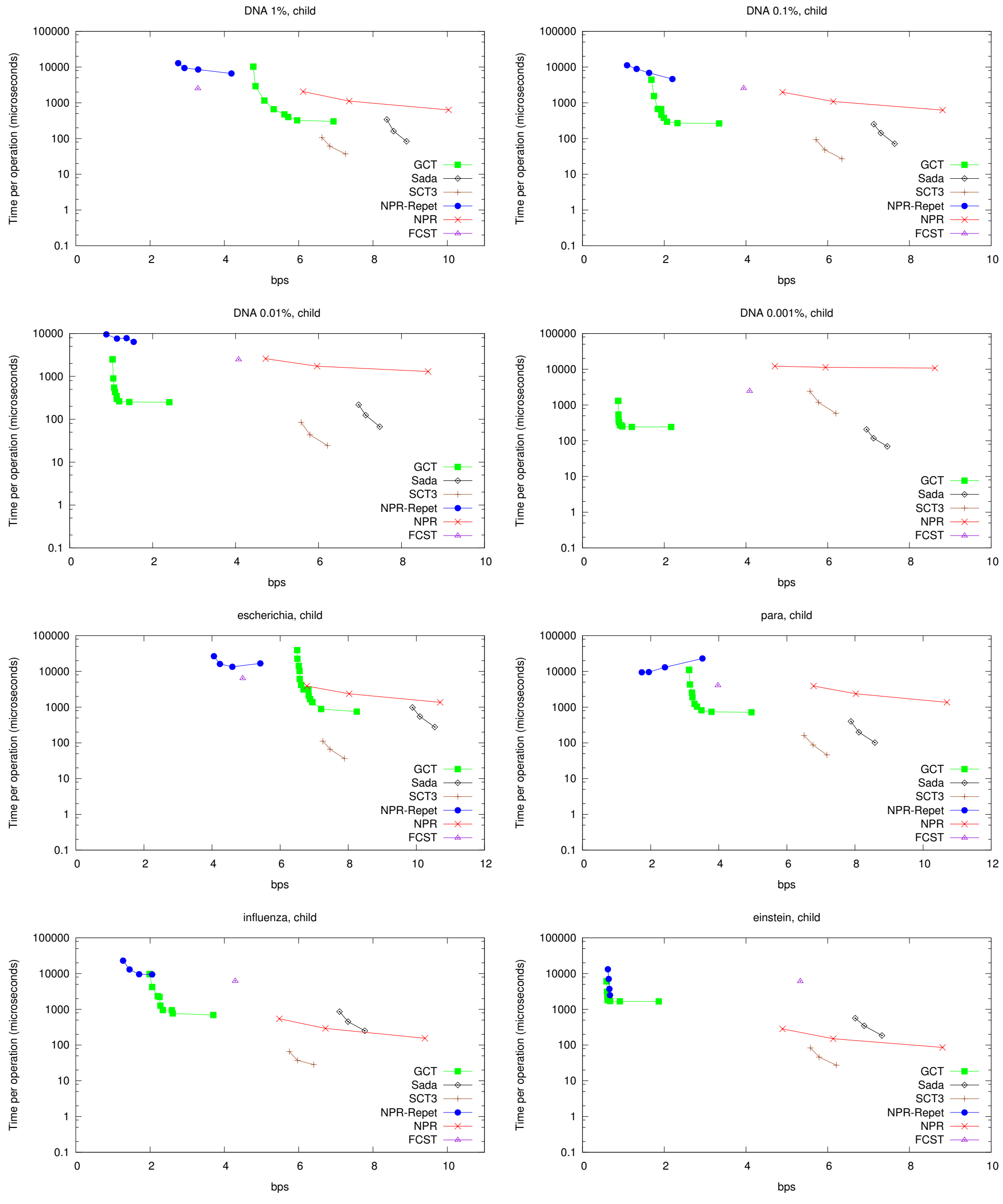


Fig. 18: Space-time tradeoffs for operation *child*.

Operation	Time (μs)	NPR-Repet	FCST	NPR	SCT3	Sada
fChild	3–10	2–3	2–3	2	(1)	(2–3)
tDepth	5–10	2–4	3–4	2–4	0–1	(2)
nSibling	10–50	2	2	1	(1–2)	(1–2)
parent	10–40	2	2	1	(1)	(3–4)
tAncestor	10–30	3–4	3–4	2–3	1–2	(1–2)
LCA	30–100	1–2	(1)	1	(1)	(1)
sLink	60–300	1–2	0–1	0–1	(1–2)	(1–2)
sDepth	50–100	0–1	1	0	0	0
sAncestor	250–700	0–1	2–3	0	0–1	0
letter	4–10	0	0	(1)		1
child	300–1000	1–2	1	0–1	(0–2)	(0–1)

Table II: Operation time ranges for the GCT and orders of magnitude of difference with alternative CSTs (the other structure is slower by that order, unless the number is in parentheses, in which case it is faster by that order). The space increases left to right, in general terms.

without the explicit topology; and *textPos* depends exclusively on the performance of the underlying CSA (albeit the GCT requires also counting leaves). Essentially, the cost of *sDepth* is that of a *nSibling* plus a *textPos* operation.

4.3. Discussion

4.3.1. Operation times. Table II shows the ranges of the operation times of the GCT over all the collections tested, and how many orders of magnitude are those times lower or higher than the competitor structures[9].

The differences are particularly striking on the operations that directly refer to the tree topology: even when the GCT significantly compresses the topology, which entails a time cost of 1–4 orders of magnitude compared to less compressed representations (Sada), this is still 1–4 orders of magnitude faster than alternative schemes, which use the topology in implicit form (an exception is the *LCA* operation on the FCST, which is very fast). On SCT3, which uses speedup structures that are alternatives to the topology, the differences in speed are up to 2 orders of magnitude in either direction, depending on the operation.

The difference decreases to 0–2 orders of magnitude on the operations that involve interaction with the CSA, as this usage is common to all the CSTs and encompasses a significant part of the total time. The general trend, within these lower gaps, is maintained: the GCT is faster than NPR-Repet, FCST and NPR, the comparison is mixed with SCT3, and Sada is faster. Exceptions are *tAncestor* on FCST, which is 2–3 orders of magnitude slower than GCT, and *letter*, which is faster on NPR than on GCT, and slower on Sada than on GCT.

The effect can also be seen on the absolute operation times. While the GCT uses 5–50 μs on direct tree operations (except on the *LCA*, which is by far the most complex one), the times raise to the range 50–1000 μs on the more complex operations that interact with the CSA. Incidentally, the *LCA* is the only operation where another structure within a competitive space range, the FCST, is faster than the GCT (other less important ones would be *isLeaf* and *ancestor*).

[9]For the sake of generalization, we omitted the 10-times faster times of NPR on `Influenza` and `Einstein`, and a couple of excessively high times of SCT3.

4.3.2. Times on a complex process. While the operation-wise comparison gives us a fine-grained picture of the performance differences, it may be difficult to determine how will the time differences look along a whole process formed by various operations of different kinds. To give a significant example of the differences between GCT and its fastest competitor, Sada, on a real-life problem, we choose a paradigmatic example of suffix tree functionality: find the maximal substrings of a new string $S[1, m]$ that are also substrings of T.

The algorithm is as follows: We descend by the suffix tree with the symbols of $S[1, i]$ until descending further is not possible. Then $S[1, i]$ is a maximal substring. Then we take the suffix link, corresponding to $S[2, i]$, and try to descend further. If this is still not possible, we keep traversing suffix links until we reach a node representing $S[j, i]$ from where it is possible to descend, until the node representing $S[j, i']$. Then $S[j, i']$ is the second maximal substring, and so on. The total process requires $O(m)$ operations *child* and *sLink*, which are among the most important ones on suffix trees.

The process, however, is complicated by the fact that the involved suffix tree nodes may not be explicit. Those *virtual* nodes are written as (v, ℓ), meaning the ℓth child along the unary path that descends from v in the suffix trie ($\ell = 0$ for explicit nodes). To take the suffix link of a virtual node, we can take the suffix link $v' = \textit{sLink}(v)$ and descend up to ℓ times from v' (as there may be some intermediate explicit nodes below v' before reaching the suffix link of (v, ℓ)). This amortizes to $O(m)$ operations, but it makes repeated use of *child*, which is one of the slowest operation for all CSTs (around 1 *ms* in the GCT and Sada). Instead, we take advantage of the faster *sAncestor* operation (around 300 μs in both CSTs) and proceed otherwise: we take the explicit descendant u of (v, ℓ), compute $u' = \textit{sLink}(u)$, and finally the desired node is $\textit{sAncestor}(u', d)$, where $d = \textit{sDepth}(u) - \textit{sDepth}(v) - \ell$ (operation $\textit{sDepth}(u)$ takes less than 100 μs in both CSTs, whereas $\textit{sDepth}(v)$ is known from the previous operation). This of course takes also $O(m)$ operations, which require less than half the time of the classical alternative.

Overall, the operations *child*, *sLink*, *sAncestor* and *sDepth* are involved. These in turn make use of the primitives *fwd*, counting leaves up to a position, computing Ψ, finding a given leaf, *LCA*, computing *LCP* values (and thus locating a suffix array position, which depends on the sampling, and accessing bitvector H), binary searching on *tAncestor* (which makes use of *bwd*), traverse the children of a node, and computing *letter* (which again applies *fwd* and Ψ). Therefore the test on the suffix tree operations is rather comprehensive.

We take `Influenza` as our text collection. For the string S, we take other Influenza sequences[10]. We take one sequence of 3000 base pairs, so the process simulates finding zones of the collection that are highly similar to a new gene. The resulting maximal intervals have lengths around 100. To consider a longer string S, we also concatenate 2 MB of those sequences (removing separators), which is the approximate length of a genome in our collection. For GCT, we take the sweet point at $y = 1$ and $z = 2^{10}$, and use the RLCSA samples of 32 for Ψ and 64, 128, and 256 for the text sampling. The RLCSA sampling used for Sada is the same.

Figure 19 shows the results. The differences between GCT and Sada are more noticeable as the RLCSA sampling is denser, since the other operations take more relevance. However, for reasonable sampling values, the differences in time are below a factor of 3. More importantly, the GCT can reach the same time performance of Sada while using much less space. For example, for the short string S, the GCT speeds up to 300 μs per symbol while using around 3 bps, so the whole process takes less than a second. If allowed to use that time, however, Sada still cannot use less than 7 bps. The

[10]From `www.cs.helsinki.fi/group/suds/rlcsa/data/influenza.gz`

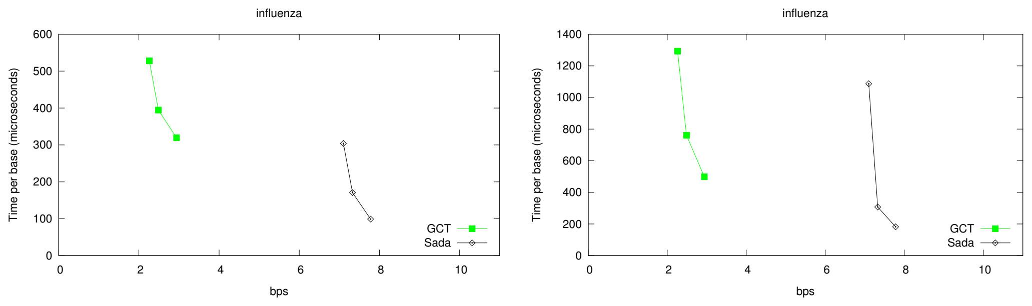


Fig. 19: Space-time tradeoffs for finding the maximal substrings of $S[1, m]$ that appear in the collection. On the left, $m = 3000$, on the right, $m = 2$MB.

differences are higher on the longer S, where the GCT can process the whole genome in around 15 minutes using 3 bps.

4.3.3. Evolution of space usage. The times obtained on larger CSTs are, of course, lower. For example, the large NPR structure [Abeliuk et al. 2013] reaches 1 μs in most operations (except 10 μs on *LCA* and 100 μs on *child*). However, as explained, it would be particularly large on repetitive collections. Other structures implemented in SDSL [Gog 2015] are larger than SCT3 and faster. In particular, the original structure by Sadakane [2007], as implemented in SDSL, should use around 9–10 bps with a sampling sufficiently dense to solve all the operations in 1–10 μs (and the direct tree operations in nanoseconds, as shown in our experiments).

Those general-purpose suffix trees will maintain their bps value approximately stable as the collection grows, whereas those oriented to repetitiveness like NPR-Repet and GCT are likely to keep reducing their bps. Figure 20 shows how the space of the CSTs considered evolves as repetitiveness increases on the synthetic DNA collections (where repetitiveness can be precisely measured). As discussed, the FCST is the only one that worsens with repetitiveness. With mutation rates of 1% the GCT uses less than 6 bps. Although NPR-Repet and FCST use less space, the GCT is orders of magnitude faster than them. When the mutation rate drops to 0.1%, the GCT becomes smaller and way faster than NPR and FCST, and the difference widens as the mutation rate drops. Only NPR-Repet stays more space-efficient than the GCT, but the difference decreases fast with repetitiveness (it would probably almost disappear at 0.001%). Still, the GCT is several orders of magnitude faster than NPR-Repet. SCT3 and Sada are faster than GCT for many operations, but already for a mutation rate of 0.1% they use more than 3 times the space of GCT. This raises to more than 5 times for the mutation rate 0.01%.

4.3.4. Construction. Finally, let us consider construction times. Figure 21 shows the cost to build the GCT separated by collections (with the average at the end) and by subprocess in the construction: from bottom to top, the construction of the RLCSA, the construction of the LCP (bitvector H), the generation of the parentheses topology, and finally its Repair-compression. This last step takes a significant portion of the total construction time, and renders the GCT 2–5 times slower to build than the classical CSTs (except the FCST, which builds more than 10 times slower). Analogously, the RePair-compression of the differential LCP array is what makes NPR-Repet equally slow to build. Still, the construction of the GCT for a human genome should take less than 2.5 hours, which seems acceptable.

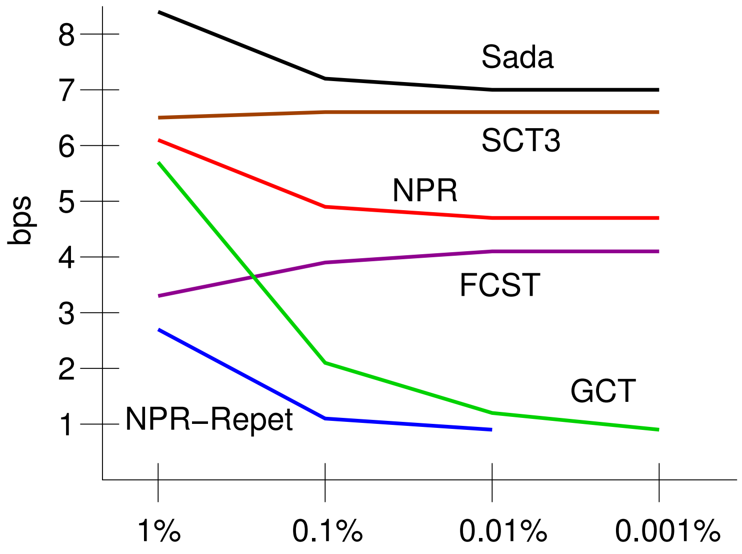


Fig. 20: Approximate space figures for the different CSTs as repetitiveness increases on the synthetic collections. For the space of GCT we take the sweet point, whereas for the others we show their minimum space in the plots.

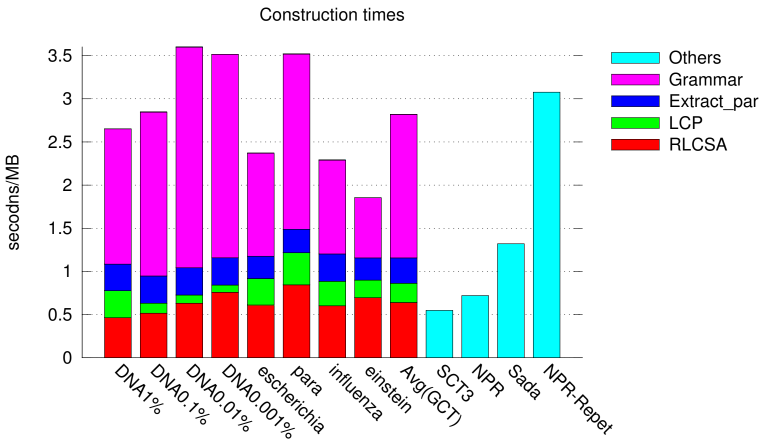


Fig. 21: Construction times, in seconds per MB, for the different indexes. The GCT is separated into the different subprocesses. The time of FCST is over 30 seconds per MB.

5. CONCLUSIONS

We have introduced the *grammar compressed tree (GCT)* a representation of arbitrary tree topologies that exploits repetitiveness, that is, identical subtrees, in a way that full navigation functionality is retained. In fact, any operation that can be solved on the sequence of parentheses [Navarro and Sadakane 2014] can also be solved on the GCT.

We have shown, in particular, that the GCT allows representing explicitly the topology of compressed suffix trees within very little space on repetitive sequence collections, using less than 2 bits per symbol (bps) for synthetic mutation rates under 0.1%, and within 2–3 bps on actual repetitive DNA sequence collections. Thanks to the explicit representation of the topology, the GCT is fast, solving the query and navigation operations in the range of the microseconds. This is generally several orders of magnitude faster than alternative representations that achieve competitive space [Abeliuk and Navarro 2012] (and not so competitive [Russo et al. 2011; Cánovas and Navarro 2010]) by managing the topology in implicit form. From those alternatives, only one [Abeliuk and Navarro 2012; Abeliuk et al. 2013] is actually smaller than the GCT (with the difference shrinking as repetitiveness increases), but its operation times are in the range of milliseconds. Only larger CSTs [Sadakane 2007; Ohlebusch et al. 2010; Cánovas and Navarro 2010; Gog 2015], which on these collections would use 3–7 times the space of the GCT, operate within microseconds and can be faster than the GCT (sometimes orders of magnitude faster) for most operations.

Needless to say, the GCT has broader interest than representing suffix tree topologies. After its conference publication [Navarro and Ordóñez 2014a], we have succeeded in using the GCT to represent the structure of versioned XML repositories [Navarro and Ordóñez 2014b]. As shown in the example of Einstein's Wikipedia article in this paper, versioned document collections may be much more repetitive than biological sequence databases. Another possible use is the representation of versioned structured software repositories.

Three important challenges remain open:

(1) Very large collections must reside on disk before they are compressed to fit in main memory. The main obstacle to handle them with our techniques is that the compression itself is not yet engineered to run on secondary memory. For example, RePair compression performs well only in main memory (but it can be replaced by other grammar compressors). This is an important future challenge in order to address massive repetitive text collections.

(2) We have been so successful in compressing the various components of the suffix tree, that the sampling of the RLCSA [Mäkinen et al. 2010], which is not compressed, starts to dominate. For example, on DNA with 0.001% of mutations, the whole GCT uses 0.9 bps, from where 0.6 bps owe to the RLCSA sampling. Finding ways to compress this sampling when the collection is repetitive is becoming a pressing issue. Some recent and promising results in this aspect point to new research directions [Na et al. 2013].

(3) Lempel-Ziv compression, especially the LZ77 variant, is more powerful than grammar compression, but more difficult to manipulate [Navarro 2012]. Further space reductions could be achieved by applying LZ77 compression, instead of RePair, to the tree topology, as long as we are able to perform the navigation operations. There is no obvious way to do it, however.

REFERENCES

ABELIUK, A., CÁNOVAS, R., AND NAVARRO, G. 2013. Practical compressed suffix trees. *Algorithms 6,* 2, 319–351.

ABELIUK, A. AND NAVARRO, G. 2012. Compressed suffix trees for repetitive texts. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 7608. Springer, 30–41.

ABOUELHODA, M., KURTZ, S., AND OHLEBUSCH, E. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2,* 1, 53–86.

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

ARROYUELO, D., CÁNOVAS, R., NAVARRO, G., AND SADAKANE, K. 2010. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 84–97.

BILLE, P., LANDAU, G., RAMAN, R., SADAKANE, K., RAO, S. S., AND WEIMANN, O. 2011. Random access to grammar-compressed strings. In *Proc. 22nd Annual Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 373–389.

BRISABOA, N., LADRA, S., AND NAVARRO, G. 2013. DACs: Bringing direct access to variable-length codes. *Information Processing and Management 49,* 1, 392–404.

CÁNOVAS, R. AND NAVARRO, G. 2010. Practical compressed suffix trees. In *Proc. 9th International Symposium on Experimental Algorithms (SEA)*. LNCS 6049. Springer, 94–105.

CLAUDE, F. AND NAVARRO, G. 2010. Self-indexed grammar-based compression. *Fundamenta Informaticae 111,* 3, 313–337.

CLAUDE, F. AND NAVARRO, G. 2012. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 7608. Springer, 180–192.

COMON, H., DAUCHET, M., GILLERON, R., LÖDING, C., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 2007. *Tree Automata Techniques and Applications*. INRIA.

CRAUSER, A. AND FERRAGINA, P. 2002. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32,* 1, 1–35.

DEMENTIEV, R., KÄRKKÄINEN, J., MEHNERT, J., AND SANDERS, P. 2008. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics 12*, article 3.4.

DO, H.-H., JANSSON, J., SADAKANE, K., AND SUNG, W.-K. 2012. Fast relative Lempel-Ziv self-index for similar sequences. In *Proc. Joint International Conference on Frontiers in Algorithmics and Algorithmic Aspects in Information and Management (FAW-AAIM)*. LNCS 7285. Springer, 291–302.

FERRAGINA, P., GAGIE, T., AND MANZINI, G. 2012. Lightweight data indexing and compression in external memory. *Algorithmica 63,* 3, 707–730.

FERRAGINA, P. AND GROSSI, R. 1999. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM 46,* 2, 236–280.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed representations of sequences and full-text indexes. *ACM TALG 3,* 2, article 20.

FISCHER, J. 2010. Wee LCP. *Information Processing Letters 110*, 317–320.

FISCHER, J., MÄKINEN, V., AND NAVARRO, G. 2009. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science 410,* 51, 5354–5364.

GAGIE, T., GAWRYCHOWSKI, P., KÄRKKÄINEN, J., NEKRICH, Y., AND PUGLISI, S. J. 2012. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*. LNCS 7183. Springer, 240–251.

GOG, S. 2015. Succinct data structures library (sdsl). `https://github.com/simongog/sdsl-lite`.

GOG, S., MOFFAT, A., CULPEPPER, J. S., TURPIN, A., AND WIRTH, A. 2014. Large-scale pattern search using reduced-space on-disk suffix arrays. *IEEE Transactions on Knowledge and Data Engineering 26,* 8, 1918–1931.

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

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th Symposium on Foundations of Computer Science (FOCS)*. IEEE, 549–554.

JORDE, L. B. AND WOODING, S. P. 2004. Genetic variation, classification and 'race'. *Nature Genetics 36,* 11s, S28–33.

KÄRKKÄINEN, J. AND KEMPA, D. 2014a. Engineering a lightweight external memory suffix array construction algorithm. In *Proc. 2nd International Conference on Algorithms for Big Data (ICABD)*. CEUR, 53–60.

KÄRKKÄINEN, J. AND KEMPA, D. 2014b. LCP array construction in external memory. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. LNCS 8504. Springer, 412–423.

KÄRKKÄINEN, J. AND RAO, S. 2003. *Algorithms for Memory Hierarchies*. LNCS 2625. Springer, Chapter 7: Full-text indexes in external memory, 149–170.

KREFT, S. AND NAVARRO, G. 2013. On compressing and indexing repetitive sequences. *Theoretical Computer Science 483*, 115–133.

KURTZ, S. 1999. Reducing the space requirements of suffix trees. *Software Practice and Experience 29,* 13, 1149–1171.

KURUPPU, S., PUGLISI, S. J., AND ZOBEL, J. 2011. Optimized relative Lempel-Ziv compression of genomes. In *Proc. 34th Australasian Computer Science Conference (ACSC)*. CRPIT 113. Australian Computer Society, 91–98.

LARSSON, J. AND MOFFAT, A. 2000. Off-line dictionary-based compression. *Proc. of the IEEE 88,* 11, 1722–1732.

LOHREY, M., MANETH, S., AND MENNICKE, R. 2011. Tree structure compression with repair. In *Proc. Data Compression Conference (DCC)*. IEEE CS, 353–362.

MÄKINEN, V., NAVARRO, G., SIRÉN, J., AND VÄLIMÄKI, N. 2010. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology 17,* 3, 281–308.

MANBER, U. AND MYERS, E. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

MANETH, S. AND BUSATTO, G. 2004. Tree transducers and tree compressions. In *Proc. 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. LNCS 2987. Springer, 363–377.

MANZINI, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

MCCREIGHT, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 32,* 2, 262–272.

MUNRO, J., RAMAN, R., RAMAN, V., AND RAO, S. S. 2003. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages, and Programming (ICALP)*. LNCS 2719. Springer, 345–356.

NA, J. C., PARK, H., CROCHEMORE, M., HOLUB, J., ILIOPOULOS, C. S., MOUCHARD, L., AND PARK, K. 2013. Suffix tree of alignment: An efficient index for similar data. In *Proc. International Workshop on Combinatorial Algorithms (IWOCA)*. LNCS 8288. Springer, 337–348.

NAVARRO, G. 2012. Indexing highly repetitive collections. In *Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA)*. LNCS 7643. Springer, 274–279.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

NAVARRO, G. AND ORDÓÑEZ, A. 2014a. Faster compressed suffix trees for repetitive text collections. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. LNCS 8504. Springer, 424–435.

NAVARRO, G. AND ORDÓÑEZ, A. 2014b. Grammar compressed sequences with rank/select support. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 8799. Springer, 31–44.

NAVARRO, G., PUGLISI, S., AND VALENZUELA, D. 2011. Practical compressed document retrieval. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*. LNCS 6630. Springer, 193–205.

NAVARRO, G. AND SADAKANE, K. 2014. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms 10,* 3, article 16.

OHLEBUSCH, E. 2013. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag.

OHLEBUSCH, E., FISCHER, J., AND GOG, S. 2010. CST++. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 6393. Springer, 322–333.

RUSSO, L., NAVARRO, G., AND OLIVEIRA, A. 2011. Fully-compressed suffix trees. *ACM Transactions on Algorithms 7,* 4, article 53.

SADAKANE, K. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems 41,* 4, 589–607.

SAKAMOTO, H. 2005. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms 3*, 416–430.

TABEI, Y., TAKABATAKE, Y., AND SAKAMOTO, H. 2013. A succinct grammar compression. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 7922. Springer, 235–246.

TISHKOFF, S. A. AND KIDD, K. K. 2004. Implications of biogeography of human populations for 'race' and medicine. *Nature Genetics 36,* 11s, S21–27.

UKKONEN, E. 1995. Constructing suffix trees on-line in linear time. *Algorithmica 14,* 3, 249–260.

WEINER, P. 1973. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*. 1–11.