Fully-Compressed Suffix Trees

LUÍS M. S. RUSSO , INESC-ID / Instituto Superior Técnico, Tech Univ of Lisbon, Portugal
GONZALO NAVARRO, University of Chile
ARLINDO L. OLIVEIRA , INESC-ID / Instituto Superior Técnico, Tech Univ of Lisbon, Portugal

Suffix trees are by far the most important data structure in stringology, with a myriad of applications in fields like bioinformatics and information retrieval. Classical representations of suffix trees require $\Theta(n \log n)$ bits of space, for a string of size n. This is considerably more than the $n \log_2 \sigma$ bits needed for the string itself, where σ is the alphabet size. The size of suffix trees has been a barrier to their wider adoption in practice. Recent compressed suffix tree representations require just the space of the compressed string plus $\Theta(n)$ extra bits. This is already spectacular, but the linear extra bits are still unsatisfactory when σ is small as in DNA sequences. In this paper we introduce the first compressed suffix tree representation that breaks this $\Theta(n)$-bit space barrier. The Fully Compressed Suffix Tree (FCST) representation requires only sublinear space on top of the compressed text size, and supports a wide set of navigational operations in almost logarithmic time. This includes extracting arbitrary text substrings, so the FCST replaces the text using almost the same space as the compressed text. An essential ingredient of FCSTs is the lowest common ancestor (LCA) operation. We reveal important connections between LCAs and suffix tree navigation. We also describe how to make FCSTs dynamic, *i.e.*, support updates to the text. The dynamic FCST also supports several operations. In particular it can build the static FCST within optimal space and polylogarithmic time per symbol. Our theoretical results are also validated experimentally, showing that FCSTs are very effective in practice as well.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Text processing, Pattern matching, String algorithms, Suffix tree, Data compression, Compressed index

ACM Reference Format:
ACM Trans. Algor. V, N, Article A (January YYYY), 33 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

First and third authors supported by FCT through projects TAGS PTDC/EIA-EIA/112283/2009, HELIX PTDC/EEA-ELC/113999/2009 and the PIDDAC Program funds (INESC-ID multiannual funding). Second author partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.
Preliminary partial versions of this paper appeared in *LATIN 2008*, LNCS 4957, pp. 362–373; and *CPM 2008*, LNCS 5029, pp. 191–203.
Authors' address: Luís M. S. Russo, Arlindo Oliveira, Instituto de Engenharia de Sistemas e Computadores: Investigação e Desenvolvimento (INESC-ID), R. Alves Redol 9, 1000-029 LISBON, PORTUGAL
Instituto Superior Técnico Technical University of Lisbon (IST/UTL), Av. Rovisco Pais, 1049-001 LISBON, PORTUGAL {lsr,aml}@kdbio.inesc-id.pt.
Gonzalo Navarro, Dept. of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

1. INTRODUCTION AND RELATED WORK

Suffix trees are extremely important for a large number of string processing problems. Many of their virtues were described by Apostolico [1985] and Gusfield [1997]. The combinatorial properties of suffix trees have a profound impact in the bioinformatics field, which needs to analyze long strings of DNA and proteins with no predefined boundaries. This partnership has produced several important results, but it has also exposed the main shortcoming of suffix trees: Their large space requirements, together with their need to operate in main memory, renders them inapplicable in the cases where they would be most useful, that is, on large texts.

There has been much research around the space problem of suffix trees, ranging from space-engineered representations [Giegerich et al. 2003; Abouelhoda et al. 2004] to novel data structures simulating them, most notably suffix arrays [Manber and Myers 1993]. Some of those space-reduced variants give away some functionality in exchange. For example suffix arrays miss the important *suffix link* navigational operation. Still, these classical approaches require $\Theta(n \log n)$ bits, while the indexed string requires only $n \log \sigma$ bits[1], n being the size of the string and σ the size of the alphabet. For example, storing the human genome requires about 700 Megabytes, while even a space-efficient suffix tree of it requires at least 40 Gigabytes [Sadakane 2007], and the reduced-functionality suffix array requires more than 10 Gigabytes. This problem is particularly evident in DNA because $\log \sigma = 2$ is much smaller than $\log n \approx 30$.

These representations are also much larger than the size of the *compressed* string. Recent approaches [Navarro and Mäkinen 2007] combining data compression and succinct data structures have achieved spectacular results for the *pattern search* problem, that is, finding the occ occurrences of a pattern string P in the text. For example, Ferragina et al. [2007] presented a compressed suffix array that, for moderate $\sigma = O(\text{polylog}(n))$ requires $nH_k + o(n)$ bits of space and computes occ in time $O(|P|)$.[2] Here nH_k denotes the k-th order empirical entropy of the string [Manzini 2001], a lower bound on the space achieved by any compressor using k-th order modeling. As that index is also able of reproducing any text substring, its space is asymptotically optimal in the sense that no k-th order compressor can achieve asymptotically less space to represent the text.

It turns out that it is possible to use this kind of data structures, that we will call *compressed suffix arrays* (CSAs)[3] and, by adding a few extra structures, support all the operations provided by suffix trees. Sadakane [2007] presented the first such *compressed suffix tree* (CST), adding $6n$ bits on top of the CSA.

In this paper we break this $\Theta(n)$ extra-bit space barrier. We use a new suffix tree representation on top of a compressed suffix array, so that we can support all of the navigational operations within $o(n)$ bits, besides the compressed suffix array, provided $\log \sigma = o(\frac{\log n}{\log \log n})$. Hence we name the data structure *Fully Compressed Suffix Tree (FCST)*. Our central tools are a particular sampling of suffix tree nodes, its connection with the suffix link and the lowest common ancestor (LCA) operation, and the interplay with the compressed suffix array. We exploit the relationship between these actors and uncover some properties that might be of independent interest.

A comparison between Sadakane's CST and our FCST is shown in Table I, considering a moderate alphabet size $\sigma = O(\text{polylog}(n))$ (there are several more operations that are trivially supported, see the end of Section 2.1). The table assumes that the CST uses the CSA of Grossi et al. [2003] (a variant that requires $(1+\frac{1}{\epsilon})nH_k + \Theta(n)$ bits

[1]In this paper $\log$ stands for $\log_2$.
[2]For general $\sigma = o(n)$, the space is $nH_k + O(\frac{n \log \sigma \log \log n}{\log n})$ and the time is $O(|P|(1 + \frac{\log \sigma}{\log \log n}))$.
[3]These are also called compact suffix arrays, FM-indexes, etc. in the literature [Navarro and Mäkinen 2007].

Table I. Comparison between compressed suffix tree representations. The operations are defined along Section 2.1 and are separated in a first group of general tree navigation and a second specific of suffix trees. The instantiation we show assumes $\sigma = O(\mathrm{polylog}(n))$, and uses different versions of the CSA of Grossi et al. for the CST and EBST, and the FM-index of Ferragina et al. for the FCST. The space given holds for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. The $o(n)$ space term in this instantiation is $O(n/\log\log n)$. CST and EBST times should be multiplied by a low-degree polynomial of $\log\log n$, which we omit for simplicity as it would be dominated by using an infinitesimally larger ϵ.

	CST	EBST	FCST
Space in bits	$(1+\frac{1}{\epsilon})nH_k + \mathbf{\Theta}(\mathbf{n})$	$(1+\frac{1}{\epsilon})nH_k + o(n)$	$nH_k + o(n)$
Root	$O(1)$	$O(1)$	$O(1)$
Count	$O(1)$	$O(1)$	$O(1)$
Ancestor	$O(1)$	$O(1)$	$O(1)$
Parent	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
FChild	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
NSib	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
LCA	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
TDep	$O(1)$	*Not supported*	$O((\log n \log\log n)^2)$
TLAQ	$O(1)$	*Not supported*	$O((\log n \log\log n)^2)$
Letter(v,i,ℓ)	$O(\log^\epsilon n + \ell/\log_\sigma n)$	$O(\log^\epsilon n + \ell/\log_\sigma n)$	$O(\log n \log\log n + \ell)$
Child	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n (\log\log n)^2)$
Locate	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
SLink	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
SLink^i	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
WeinerLink	$O(\log n)$	$O(\log n)$	$O(1)$
SDep	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
SLAQ	$O(\log^{1+\epsilon} n)$	$O(\log^{1+\epsilon} n)$	$O(\log n \log\log n)$

for any constant $\epsilon > 0$) and that the FCST uses the FM-index [Ferragina et al. 2007] (which requires $nH_k + o(n)$ bits), to take the preferred setting for each. In general the FCST is slower than the CST, but it requires much less space. Assuming realistically that for DNA $H_k \approx 2$, Sadakane's CST requires at the very least $8n + o(n)$ to $13n + o(n)$ bits, depending on the CSA variant of Grossi et al. [2003] they use, whereas the FCST requires only $2n + o(n)$ bits (this theoretical prediction is not far from reality, as shown in Section 7). The FCST space is optimal in the sense that no k-th order compressor can achieve asymptotically less space to represent T. If the CST used the FM-index, it would still have the $6n$ extra bits and the $O(\log^\epsilon n)$ time complexities would become $O(\log n \log\log n)$.

Table I also compares the *Entropy-Bounded Suffix Tree (EBST)* [Fischer et al. 2009; Fischer 2010], a newer proposal that aims at maintaining the $o(n)$ extra space of the FCST while reducing navigation times. If it uses another version of the CSA by Grossi et al. [2003] that requires $o(n)$ extra bits on polylog-sized alphabets, it achieves sublogarithmic time complexities for most operations. If we force it to use the FM-index to achieve the least possible space (as the FCST) its time complexities become not competitive. There are previous incomplete theoretical proposals for compressed suffix trees [Munro et al. 2001; Foschini et al. 2006]; a brief description is given at the end of Section 3.

Our results are based on a special kind of sampling of suffix tree nodes. There is some literature on sampled, or sparse, suffix trees. The pioneering work [Kärkkäinen and Ukkonen 1996b] indexed evenly spaced suffixes (every k text positions). The resulting structure required reduced space, $O((n/k)\log n) + n\log\sigma$ bits, at the price of multiplying the suffix tree search time by k and only handling patterns of length k or more. Replacing the regular sampling with one guided by the Lempel-Ziv parsing yielded the very first compressed text index [Kärkkäinen and Ukkonen 1996a]. This index used the Lempel-Ziv properties to handle any pattern length, and later several

Table II. Comparison between dynamic compressed suffix tree representations. The operations are defined along Section 2.1. The same considerations of Table I apply, except that the instantiation assumes the dynamic FM-Index variant of Navarro and Sadakane [2010] as the CSA, for which the space holds for any $k \leq \alpha \log_\sigma(n) - 1$ and any constant $0 < \alpha < 1$.

	Chan et al. [2007] (DCST)	Ours (DFCST)
Space in bits	$nH_k + \mathbf{\Theta(n)}$	$nH_k + o(n)$
ROOT	$O(1)$	$O(1)$
COUNT	$O(\log n/\log\log n)$	$O(1)$
ANCESTOR	$O(\log n/\log\log n)$	$O(1)$
PARENT	$O(\log n/\log\log n)$	$O(\log^2 n)$
FCHILD	$O(\log n/\log\log n)$	$O(\log^2 n \log\log n)$
NSIB	$O(\log n/\log\log n)$	$O(\log^2 n \log\log n)$
LCA	$O(\log n/\log\log n)$	$O(\log^2 n)$
LETTER(v, i, ℓ)	$O(\log n(\log n + \ell/\log\log n))$	$O(\log n(\log n + \ell/\log\log n))$
CHILD	$O(\log^2 n \log\sigma)$	$O(\log^2 n \log\log n)$
LOCATE	$O(\log^2 n)$	$O(\log^2 n)$
SLINK	$O(\log n/\log\log n)$	$O(\log^2 n)$
SLINK^i	$O(\log^2 n)$	$O(\log^2 n)$
WEINERLINK	$O(\log n/\log\log n)$	$O(\log n/\log\log n)$
SDEP	$O(\log^2 n)$	$O(\log^2 n)$
INSERT(T) / DELETE(T)	$O(\lvert T\rvert \log^2 n)$	$O(\lvert T\rvert \log^2 n)$

self-indexes based on Lempel-Ziv compression followed the same lines [Navarro 2004; Ferragina and Manzini 2005; Russo and Oliveira 2008]. Sparse indexes that use evenly spaced suffixes and orthogonal range searching were recently proposed for secondary memory searching [Chien et al. 2008; Hon et al. 2009]. All these representations support pattern searches, but not the full suffix tree functionality. Our sampling is different in the sense that it samples suffix tree nodes, not text positions. This is the key to achieve good upper bounds for all suffix tree operations.

Albeit very appealing, static FCSTs must be built from the uncompressed suffix tree. Moreover, they must be rebuilt from scratch upon changes in the text. This severely limits their applicability, as one needs to have a large main memory, or resort to secondary memory construction, to end up with a FCST that fits in a reasonable main memory. CSAs have overcome this limitation, starting with the structure by Chan et al. [2004]. In its journal version [Chan et al. 2007] the work includes the first dynamic CST, which builds on the static CST of Sadakane [2007] and retains its $\Theta(n)$ extra space penalty (with constant at least 6). On the other hand, the smallest existing CSA [Ferragina et al. 2007] was made dynamic within the same space by Navarro and Sadakane [2010] so as to achieve a sublogarithmic slowdown with respect to the static version[4]. In this paper we show how to support dynamic FCSTs, by building on this latter dynamic CSA. We retain the optimal space complexity and polylogarithmic time for all the operations.

A comparison between the dynamic CST by Chan et al. [2007] and our dynamic FCST is given in Table II. Both use the dynamic FM-index of Navarro and Sadakane [2010], as that of Chan et al. [2007] uses $O(\sigma n)$ space and is not significantly faster. Again, the FCST is slower but requires much less space (one can realistically predict 25% of Chan et al.'s CST space on DNA).

All these dynamic structures, as well as ours, handle a *collection* of texts, where whole texts are added/deleted to/from the collection. Construction in compressed space is achieved by inserting a text into an empty collection.

[4]He and Munro [2010] obtained a very similar result but their $o(n)$ extra space term is larger, $O(n \log\log n/\sqrt{\log n})$ versus $O(n \log\log n/\log n)$.

We have implemented the static FCST and compared with an implementation of Sadakane's CST by Välimäki et al. [2007]. Our experiments show that we can obtain very small FCST representations and still support the usual operations efficiently.

We start with Section 2 by defining basic concepts about suffix trees and compact data structures, and listing the navigational operations we wish to support. In Section 3 we overview compressed suffix arrays (CSAs) and their functionality. Section 4 introduces our sampled suffix tree and shows how to support its navigation and mapping from the full (not represented) suffix tree. Section 5 shows how to carry out navigational operations using self-similarity properties of suffix trees; a detailed analysis of these properties is given in Appendix 9. Section 6 introduces our main technique for maintaining the sampling up to date upon changes in the text collection, and obtains the results on dynamic FCSTs. Section 7 shows experimental results. Section 8 considers different sampling factors to obtain larger, but faster representations. Section 9 concludes the paper.

2. BASIC CONCEPTS

In this section we give a brief review of suffix trees, suffix arrays, and compact data structures. For a more detailed explanation, the reader is referred to the publications focused on the subject, e.g. [Gusfield 1997; Navarro and Mäkinen 2007]. In particular, the former reference shows dozens of algorithms relevant in bioinformatics where the suffix tree navigation operations we are going to describe are of use.

2.1. Strings, Trees, Suffix Trees and Arrays

We denote by $T = T[0, n-1]$ a **string**, which is a sequence of **length** $|T| = n$ over an **alphabet** Σ of size σ. We denote by $T[i]$ the symbol at position $(i \bmod n)$; by $T[i..j]$ the **substring** $T[i]T[i+1] \ldots T[j]$, which is a **prefix** if $i = 0$ (and can be written $T[..j]$) and a **suffix** if $j = n-1$ (and can be written $T[i..]$). By $T.T'$ we denote the **concatenation** of T and T'. The **empty string** of length zero is denoted ε.

We make extensive use of rooted trees. The root node is called $\textsc{Root}$. By $\textsc{Parent}(v)$ we denote the parent node of node $v \neq \textsc{Root}$; by $\textsc{TDep}(v)$ its tree-depth; by $\textsc{Height}(v)$ the distance between v and its farthest descendant leaf; by $\textsc{FChild}(v)$ its first child, if v is not a leaf; and by $\textsc{NSib}(v)$ the next child of the same parent, if it exists. By $\textsc{Ancestor}(v, v')$ we denote whether v is an **ancestor** of v'; by $\textsc{LCA}(v, v')$ the **lowest common ancestor** of v and v'; and by $\textsc{TLAQ}(v, d)$ the **level-d ancestor** of v (that is, the ancestor of v with tree-depth d).

A **compact tree** is a tree that has no unary nodes (that is, nodes with only one child). A **labeled tree** is a tree that has a nonempty string **label** for every edge. In a **deterministic tree**, the common prefix of any two different edges out of a node is ε.

Definition 2.1. [Weiner 1973; McCreight 1976] The **suffix tree** $\mathcal{T}$ of a text string T is the deterministic compact labeled tree for which the path labels of the leaves are the suffixes of $T\$$, where \$ is a terminator symbol not belonging to Σ. We will assume n is the length of $T\$$.

Figure 1 shows a running example that illustrates several concepts of suffix trees, for $T = abbbaab$. The suffix tree $\mathcal{T}$ contains t nodes, and it holds $n \leq t < 2n$. In a deterministic tree the first letters of every edge are referred to as **branching letters**. A **point** p in a labeled tree is either a node or a string position in some edge label. The **path-label** of a point p in a labeled tree is the concatenation of the edge labels from the $\textsc{Root}$ down to p. *We refer indifferently to nodes v and to their path labels, also denoted by v.* The i-th letter of the path label of node v is denoted $\textsc{Letter}(v, i) = v[i]$, and in general we use $\textsc{Letter}(v, i, \ell) = v[i..i + \ell - 1]$. The **string-depth** of a node v, denoted $\textsc{SDep}(v)$, is the length of its path label. $\textsc{SLAQ}(v, d)$ is the highest ancestor v' of node v with $\textsc{SDep}(v') \geq d$. $\textsc{Child}(v, X)$ is the node that results of descending

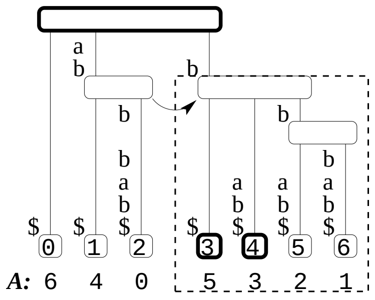

Fig. 1. Suffix tree $\mathcal{T}$ of string $abbbab$, with the leaves numbered. The arrow shows the SLINK between node ab and b. Below we show the suffix array. The portion of the tree corresponding to node b and respective leaves interval is within a dashed box. The sampled nodes have bold outlines.

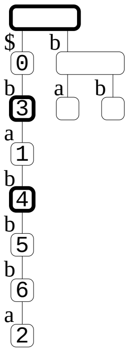

Fig. 2. Reverse tree $\mathcal{T}^R$.

```
                1             2
i: 01 234 56 7890 12 345 67 8901
   ((0)((1)(2))((3)(4)((5)(6))))

B: 1 0   0  0   101101  0  0   1
   ( 0   1  2   (3)(4)  5  6   )
i: 0            1 23 4         5

B: 1 0   0  0  1101101  0  0  11
   ( 0   1  2  ((3)(4)  5  6  ))
i: 0           12 34 5        67
```

Fig. 3. Parentheses representations of trees. The parentheses on top represent the suffix tree, those in the middle the sampled tree, and those on the bottom the sampled tree when b is also sampled along with the B bitmap. The numbers are not part of the representation; they are shown for clarity. The rows labeled `i:` give the index of the parentheses.

from v by the edge whose label starts with symbol X, if it exists. The **suffix-link** of a node $v \neq$ ROOT of a suffix tree, denoted SLINK(v), is a pointer to node $v[1..]$ (that is, the longest proper suffix of v; this node always exists). Note that SDEP(v) of a leaf v identifies the suffix of $T\$$ starting at position LOCATE$(v) = n -$ SDEP(v). In our example, $T[\text{LOCATE}(ab\$)..] = T[(7-3)..] = T[4..] = ab\$$. The list of LOCATE values comprises another well-known structure.

Definition 2.2. [Manber and Myers 1993] The **suffix array** $A[0, n-1]$ of a text T is the sequence of starting positions of the suffixes of $T\$$ in lexicographical order. This is the same as the LOCATE values of the suffix tree leaves, if the children of the nodes are ordered lexicographically by their branching letters.

Note that A is a permutation, and permutation $A^{-1}[j]$ gives the lexicographical rank of $T[j..]$ among all the suffixes of $T\$$.

The suffix tree nodes can be identified with suffix array intervals: Each node v corresponds to the *range* $[v_l, v_r]$ of leaves that descend from v (since there are no unary nodes, there are no two nodes with the same interval). These intervals are also refered to as lcp-intervals [Abouelhoda et al. 2004]. In our example, node b corresponds to the interval $[3, 6]$. We will refer indifferently to nodes v and to their interval $[v_l, v_r]$. Leaves v correspond to $[v, v]$ in this notation. For example by $v_l - 1$ we refer to the leaf immediately before v_l, *i.e.*, $[v_l - 1, v_l - 1]$. With this representation we can solve COUNT$(v) = v_r - v_l + 1$, the number of leaves that descend from node v. In our example, the number of leaves below b is $4 = 6 - 3 + 1$. *This is precisely the number of times the string v occurs in the text T*, and thus the pattern search problem for P reduces to navigating from the ROOT to the point denoting P, and then using COUNT to determine the number of times P occurs in T, and using LOCATE$(v_l) \ldots$ LOCATE(v_r) to output the occurrence positions.

The representation of ranges lets one trivially compute several other operations of interest for suffix trees, such as ANCESTOR$(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$, knowing whether v follows v' in $\mathcal{T}$ ($\Leftrightarrow v'_r < v_l$), whether the preorder of v is smaller than that of v' ($\Leftrightarrow v_l < v'_l \vee (v_l = v'_l \wedge v_r > v'_r)$), whether a node is a leaf ($v_l = v_r$), the leftmost leaf of node v (v_l), etc.

2.2. Compact Data Structures

We make heavy use of compact and compressed data structures for representing bitmaps and trees. We give now the results we build on along the paper. As already mentioned, we will develop a static and a dynamic variant of our FCST. Hence we will give static and dynamic variants of all the data structures we create.

Let $B[0, n-1]$ be a bitmap of length n. Then we define operations $\textsc{Rank}_b(B, i)$ as the number of occurrences of bit b in $B[0, i]$, and $\textsc{Select}_b(B, j)$ as the position of the $(j+1)$-th occurrence of bit b in B. We build on the following compressed bitmap representations that support $\textsc{Rank}$ and $\textsc{Select}$. Their space is asymptotically optimal among all the bitmaps with the same number of bits set [Pătraşcu and Viola 2010].

THEOREM 2.3 (2.3 [PĂTRAŞCU 2008]). *Let $B[0, n-1]$ contain m 1s. Then there exists a data structure that answers* $\textsc{Rank}$ *and* $\textsc{Select}$ *in constant time $O(c)$ using $m \log \frac{n}{m} + O(m + n/\log^c n)$ bits of space.*

In the dynamic scenario, we also wish to support operations $\textsc{Insert}(B, i, b)$, which inserts bit b between $B[i-1]$ and $B[i]$, and $\textsc{Delete}(B, i)$, which deletes $B[i]$ from the sequence.

THEOREM 2.4 (2.4 [NAVARRO AND SADAKANE 2010]). *Let $B[0, n-1]$ contain m 1s. Then there exists a data structure that answers* $\textsc{Rank}$ *and* $\textsc{Select}$*, and executes operations* $\textsc{Insert}$ *and* $\textsc{Delete}$*, all in $O(\log n/\log\log n)$ time, using $m \log \frac{n}{m} + O(m + \frac{n \log\log n}{\log n})$ bits of space.*

We will also handle general trees of n nodes, which can be represented using $2n+o(n)$ bits while supporting in constant time a number of traversal operations. This space is asymptotically optimal considering all the trees of n nodes. For this paper we are interested in the following operations: $\textsc{Preorder}(v)$ (and its inverse), which gives the preorder position of node v in the tree starting at zero; $\textsc{Parent}(v)$; $\text{LCA}(v, v')$, $\text{TD}\textsc{ep}(v)$ and $\text{TLAQ}(v, d)$.

A useful tree representation, which will be necessary at some points in the paper, is based on balanced parentheses: Do a preorder traversal and write a `'('` when you arrive at a node and a `')'` when you leave it. This sequence is regarded as a bitmap supporting $\textsc{Rank}$ and $\textsc{Select}$ operations. In addition, the following operations on the parentheses are supported: $\textsc{FindMatch}(u)$ finds the matching parenthesis of u; $\textsc{Enclose}(u)$ finds the nearest pair of matching parentheses that encloses u; and in some cases $\textsc{DoubleEnclose}(u, u')$, which finds the nearest pair of parentheses that encloses both u and u'.

These operations on the parentheses support most of the tree operations we need. If tree node v is identified with the position of its opening parenthesis in the sequence B, then $\textsc{Preorder}(v) = \textsc{Rank}_{'('}(B, v) - 1$, $\textsc{Preorder}^{-1}(i) = \textsc{Select}_{'('}(B, i)$, $\text{TD}\textsc{ep}(v) = \textsc{Rank}_{'('}(B, v) - \textsc{Rank}_{')'}(B, v)$, $\textsc{Parent}(v) = \textsc{Enclose}(v)$, $\text{LCA}(v, v') = \textsc{DoubleEnclose}(v, v')$. Only operation $\text{TLAQ}(v, d)$ needs special treatment. We will use a representation that supports all of these operations within optimal space [Pătraşcu and Viola 2010].

THEOREM 2.5 (2.5 [SADAKANE AND NAVARRO 2010]). *Let a general tree of n nodes be represented as a sequence of $2n$ balanced parentheses. Then there exists a data structure supporting operations* $\textsc{Preorder}(v)$*,* $\textsc{Preorder}^{-1}(i)$*,* $\text{LCA}(v, v')$*,* $\text{TD}\textsc{ep}(v)$*,* $\text{TLAQ}(v, d)$*, and* $\textsc{Parent}(v)$*, on the tree, and* $\textsc{Rank}(v)$*,* $\textsc{Select}(v)$*,* $\textsc{FindMatch}(v)$*,* $\textsc{Enclose}(v)$*, and* $\textsc{DoubleEnclose}(v, v')$ *on the parentheses, in constant time $t_{\mathsf{tree}} = O(c)$ using $2n + O(n/\log^c n)$ bits of space.*

In the dynamic case, we wish to INSERT tree nodes, and DELETE tree leaves or unary nodes. The update operations are then translated into $\text{INSERT}(u, u')$ and $\text{DELETE}(u, u')$, which insert or delete matching parentheses located at u, u'. On the other hand, we will not need $\textbf{TLAQ}(v, d)$.

THEOREM 2.6 (2.6 [NAVARRO AND SADAKANE 2010]). *A sequence of $2n$ balanced parentheses can be maintained in $2n + O(n \log \log n / \log n)$ bits of space while supporting the same operations of Theorem 2.5 except* $\textbf{TLAQ}(v, d)$*, plus* $\text{INSERT}(v, v')$ *and* $\text{DELETE}(v, v')$*, in $t_{\mathsf{tree}} = O(\log n / \log \log n)$ worst-case time.*

3. USING COMPRESSED SUFFIX ARRAYS

We are interested in compressed suffix arrays (CSAs) because they have very compact representations and support partial suffix tree functionality (being usually more powerful than the classical suffix arrays [Navarro and Mäkinen 2007]). Apart from the basic functionality of retrieving $A[i] = \text{LOCATE}(i)$, state-of-the-art compressed suffix arrays support operation $\text{SLINK}(v)$ *for leaves* v. This is called $\psi(v)$ in the literature: $A[\psi(v)] = A[v] + 1$, and thus $\text{SLINK}(v) = \psi(v)$, let its time complexity be $O(t_\psi)$. The iterated version of ψ, denoted ψ^i, can usually be computed faster than $O(i\, t_\psi)$ with compressed indexes. This is achieved with A and A^{-1}, $\psi^i(v) = A^{-1}[A[v] + i]$. Let $O(t_{\mathsf{SA}})$ be the time complexity to compute A and A^{-1} (and hence to compute LOCATE). CSAs also support operation $\text{WEINERLINK}(X, v)$, which, for a node v, gives the suffix tree node with path label $X.v$. This is called the LF mapping (for leaves) in compressed suffix arrays, and is a kind of inverse of ψ: $\textbf{LF}(X, v)$ gives the lexicographical rank of the suffix $X.T[A[v]..]$ among all the suffixes, whether it exists or not. Let $O(t_{\mathsf{LF}})$ be the time complexity to compute LF. It is easy to extend LF to suffix tree nodes v: $\textbf{LF}(X, v) = \textbf{LF}(X, [v_l, v_r]) = [\textbf{LF}(X, v_l), \textbf{LF}(X, v_r)] = \text{WEINERLINK}(X, v)$.

Consider the interval $[3, 6]$ in our example, which represents the leaves whose path labels start by b. In this case we have that $\textbf{LF}(a, [3, 6]) = [1, 2]$, *i.e.*, by using the LF mapping with a we obtain the interval of leaves whose path labels start by ab. We also extend the notation of LF to strings, $\textbf{LF}(X.\alpha, v) = \textbf{LF}(X, \textbf{LF}(\alpha, v))$.

Compressed suffix arrays are usually *self-indexes*, meaning that they replace the text: It is possible to extract any substring, of size ℓ, of the indexed text in $O(t_{\mathsf{SA}} + t_\psi(\ell - 1))$ time. A particularly easy case that is solved in constant time is to extract $T[A[v]]$ for a suffix array cell v, that is, the first letter of a given suffix. Since suffixes are lexicographically sorted, one can partition A into at most σ intervals, where suffixes start with the same letter. Self-indexes store, in some way or another, $O(\sigma \log \frac{n}{\sigma})$ bits that allow one to compute in constant time the partition where any v belongs [Navarro and Mäkinen 2007]. This corresponds to $\text{LETTER}(v, 0) = v[0]$, the first letter of the path label of leaf v. To obtain $\text{LETTER}(v, 0, \ell)$ one repeats the process ℓ times, on v, $\psi(v)$, $\psi(\psi(v))$, and so on, in $O(1 + t_\psi(\ell - 1))$ time. The more general $\text{LETTER}(v, i, \ell)$ is reduced to $\text{LETTER}(\psi^i(v), 0, \ell)$ in $O(t_{\mathsf{SA}} + t_\psi(\ell - 1))$ time. Finally, in order to extract $T[i..j]$ one computes $v = A^{-1}[i]$ and then $\text{LETTER}(v, 0, j - i + 1)$, in $O(t_{\mathsf{SA}} + t_\psi(j - i))$ time.

As anticipated, our compressed suffix tree representation will consist of a sampling of the suffix tree plus a compressed suffix array representation. A well-known compressed suffix array is that of Grossi et al. [2003]. One variant (their Lemma 3.2) requires $(1 + \frac{1}{\epsilon})nH_k + 2n(1 + \log e) + O(\frac{n \log \log n}{\log n})$ bits of space and has times $t_\psi = O(1)$, $t_{\mathsf{SA}} = O(\log_\sigma^\epsilon n + \log \sigma)$, and $t_{\mathsf{LF}} = O(t_\psi \log n)$, for any $0 < \epsilon < 1$, any $k \le \alpha \log_\sigma n$, and any constant $0 < \alpha < 1$. A second variant (their Theorem 4.1) requires $(1 + \frac{1}{\epsilon})nH_k + O(\frac{n \log \sigma \log \log n}{\log n})$ bits and has times $t_\psi = O(1 + \frac{\log \sigma}{\log \log n})$ (using multiary wavelet trees [Ferragina et al. 2007] instead of their earlier binary wavelet trees), $t_{\mathsf{SA}} = O(t_\psi \log_\sigma^\epsilon n)$,

Table III. Operations supported by static compressed suffix arrays (CSAs) and their space and time complexities, which hold for any $0 < \epsilon < 1$, $l \geq 1$, $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. We give two variants of Grossi et al. [2003].

	Time	[Grossi et al. 2003]	[Grossi et al. 2003]	[Ferragina et al. 2007]
Space in bits		$(1+\frac{1}{\epsilon})nH_k$ $+\Theta(n)$	$(1+\frac{1}{\epsilon})nH_k+$ $+O(\frac{n\log\log n}{\log_\sigma^{\epsilon/(1+\epsilon)} n})$	$nH_k + O(\frac{n\log\sigma\log\log n}{\log n})$ $+O((n/l)\log n)$
$\psi(v)$	t_ψ	$O(1)$	$O(1+\frac{\log\sigma}{\log\log n})$	$O(1+\frac{\log\sigma}{\log\log n})$
$A[v]$,$A^{-1}[v]$ LOCATE(v)	t_{SA}	$\log_\sigma^\epsilon n + \log\sigma$	$t_\psi \log_\sigma^\epsilon n$	$l\, t_{\mathsf{LF}}$
LF(v) WEINERLINK(v)	t_{LF}	$t_\psi \log n$		$O(1+\frac{\log\sigma}{\log\log n})$
$v[i..i+\ell-1]$ $T[i..i+\ell-1]$ LETTER(v,i,ℓ)		$t_{\mathsf{SA}} + \ell/\log_\sigma n$		$t_{\mathsf{SA}} + t_{\mathsf{LF}}(\ell-1)$ $= t_{\mathsf{LF}}(l+\ell-1)$

and $t_{\mathsf{LF}} = O(t_\psi \log n)$.[5] For our results we favor another compressed suffix array, called the FM-index [Ferragina et al. 2007], which requires $nH_k + O(\frac{n\log\sigma\log\log n}{\log n})$ bits, with the same limit on k. Its complexities are[6] $t_\psi = t_{\mathsf{LF}} = O(1+\frac{\log\sigma}{\log\log n})$ (using multiary wavelet trees again) and $t_{\mathsf{SA}} = O(l\, t_{\mathsf{LF}})$. This l is a suffix array sampling parameter, such that we need $O((n/l)\log n)$ extra bits of space. For example, if we set the extra space to $O(\frac{n}{\log\log n})$ then we use $l = \log n \log\log n$ and achieve $t_{\mathsf{SA}} = O(\log n(\log\sigma + \log\log n))$. Table III summarizes the supported CSA operations and times.

We remark that, if $\log\sigma = \Omega(\frac{\log n}{\log\log n})$, the extra space of the FM-index includes an extra $\Omega(n)$-bit term. Although this is still $o(n\log\sigma)$ bits, which can be argued to be reasonable for a text T whose plain representation takes $n\log\sigma$ bits, the main point of this paper is to break the $\Theta(n)$ space barrier. In this sense, our results are interesting for $\log\sigma = o(\frac{\log n}{\log\log n})$, where the FM-index takes $nH_k + o(n)$ bits of space. This is a reasonable assumption on σ and includes the interesting case $\sigma = O(\text{polylog}(n))$, on which the FM-index offers constant t_ψ and t_{LF} times.

Let us now consider dynamic CSAs. These handle a collection of texts, as if they formed a single concatenated text. They offer the same functionalities of the static CSAs on that text, plus insertion/deletion of texts into/from the collection. Two main dynamic CSAs exist: That of Chan et al. [2007] is a natural extension of the static CSA of Sadakane [2003]. It requires $O(\sigma n)$ bits of space, and offers complexities $t_\psi = t_{\mathsf{LF}} = O(\log n)$, $t_{\mathsf{SA}} = O(\log^2 n)$, and insertion/deletion of a text T in time $O(|T|(t_\psi + t_{\mathsf{LF}}))$. The FM-index has several dynamic versions as well [Ferragina and Manzini 2000; Mäkinen and Navarro 2008; González and Navarro 2008]. The most efficient version [Navarro and Sadakane 2010] achieves $nH_k + O(\frac{n\log\sigma\log\log n}{(1-\epsilon)\log^\epsilon n})$ bits of space, for any $k \leq \alpha\log_\sigma(n) - 1$ and any constants $0 < \alpha, \epsilon < 1$. It offers times $t_\psi = t_{\mathsf{LF}} = O(\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$, and $t_{\mathsf{SA}} = l\, t_{\mathsf{LF}}$ using other $O((n/l)\log n)$ extra bits. Again, we set $l = \log n\log\log n$ to achieve $O(\frac{n}{\log\log n})$ extra bits, which makes $t_{\mathsf{SA}} = O(\log^2 n(1+\frac{\log\sigma}{\log\log n}))$. Insertion and deletion of a text T takes time $O(|T|(t_\psi + t_{\mathsf{LF}}))$.[7]

[5]The complexities of both variants are incorrectly mixed in Fischer et al. [2009], an error that carries over to Fischer [2010].

[6]Function $\psi(v)$ can be computed in the same time of LF on the FM-index [Lee and Park 2007].

[7]This was just $O(|T|t_{\mathsf{LF}})$ in the original papers [Mäkinen and Navarro 2008; González and Navarro 2008], but in Section 6 we will modify the deletion procedure to operate in left-to-right order. Thus our times are $O(|T|t_{\mathsf{LF}})$ for insertions and $O(|T|t_\psi)$ for deletions.

Table IV. Operations supported by dynamic compressed suffix arrays (CSAs) and their space and time complexities, which hold for any $0 < \epsilon < 1$, $l \geq 1$, $k \leq \alpha \log_\sigma(n) - 1$ and constant $0 < \alpha < 1$.

	Time	[Chan et al. 2007]	[Navarro and Sadakane 2010]
Space in bits		$O(\sigma n)$	$nH_k + O(\frac{n \log \sigma}{(1-\epsilon) \log^\epsilon n})$ $+O((n/l) \log n)$
$\psi(v)$	t_ψ	$O(\log n)$	$O(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$
$A[v]$,$A^{-1}[v]$, $\textsc{Locate}(v)$	t_{SA}	$O(\log^2 n)$	$l\, t_{\mathsf{LF}}$
LF(v), $\textsc{WeinerLink}(v)$	t_{LF}	O$(\log n)$	O$(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$
$v[i..i+\ell-1], T[i..i+\ell-1]$ $\textsc{Letter}(v, i, \ell)$		$t_{\mathsf{SA}} + t_\psi(\ell - 1)$	$t_{\mathsf{SA}} + t_{\mathsf{LF}}(\ell - 1)$ $= (l + \ell - 1)t_{\mathsf{LF}}$
$\textsc{Insert}(T)$ / $\textsc{Delete}(T)$	$\lvert T\rvert(t_\psi + t_{\mathsf{LF}})$		

Table IV summarizes these complexities. The dynamic CSA by Chan et al. [2007] is used in his CST representation. In FCST representation the focus is on minimal, $o(n)$, space and therefore we will use the result by Navarro and Sadakane [2010].

A larger, but much faster, dynamic CSA was proposed by Gupta et al. [2007]. Their dynamic CSA requires $n \log \sigma + o(n \log \sigma)$ bits of space and supports queries in $O(\log \log n)$ time, and $O(1)$ time when $\sigma = O(\text{polylog}(n))$. Updates, however, are much more expensive, $O(n^\epsilon)$ amortized time, for $0 < \epsilon < 1$. The FCST representation may use this dynamic CSA. However, for this to be useful, one should also use faster dynamic trees. While there are some supporting various operations in time $O(\log \log n)$ [Raman and Rao 2003; Arroyuelo 2008], none of these supports the crucial LCA operation.

Finally, let us mention a previous data structure called a "compressed suffix tree" but which, under the terminology of this paper, offers just compressed suffix array functionality. Munro et al. [2001] propose what can be considered as a predecessor of Sadakane's CST, as it uses a suffix array and a compact tree. By using it on top of an arbitrary CSA, its smallest variant would take $|CSA| + o(n)$ bits plus the text (which could be compressed to $nH_k + o(n \log \sigma)$ bits and support constant-time access [Ferragina and Venturini 2007]) and find the suffix array interval corresponding to pattern $P[1, m]$ in time $O(m\, t_{\mathsf{SA}} \log \sigma)$. The FM-index alone, however, is the smallest CSA and can do the same without any other structure in time $O(m\, t_{\mathsf{LF}})$, which is always faster. Munro et al. [2001] can also achieve time $O(m\, t_{\mathsf{SA}})$, but for this they require $|CSA| + O(n \log \sigma)$ bits and still do not support any other suffix tree operation. There exists another previous compressed suffix tree description [Foschini et al. 2006] based on an interval representation and sampling of the suffix tree. However, the description is extremely brief and no details nor theoretical bounds on the result are given.

4. THE SAMPLED SUFFIX TREE

A pointer based implementation of suffix trees requires $\Theta(n \log n)$ bits to represent a suffix tree of $t < 2n$ nodes. As this is too much, we will store only a few *sampled* nodes. We denote our sampling factor by δ, so that in total we sample $O(n/\delta)$ nodes. Hence, provided $\delta = \omega(\log n)$, the sampled tree can be represented using $o(n)$ bits. To settle ideas we can assume $\delta = \lceil \log n \log \log n \rceil$. In our running example we use $\delta = 4$.

To illustrate the structure of the sampled tree, Figure 3 shows the balanced parentheses representation of the tree of Figure 1. The representation of the sampled tree is obtained by deleting the parentheses of the non-sampled nodes, as in Figure 3. For the sampled tree to be representative of the suffix tree it is necessary that every node is, in some sense, *close enough* to a sampled node.

Definition 4.1. A *δ-sampled tree* S of a suffix tree $\mathcal{T}$ with t nodes is formed by choosing $s = O(t/\delta)$ nodes of $\mathcal{T}$ so that for each node v of $\mathcal{T}$ there is an $i < \delta$ such that node $\textsc{SLink}^i(v)$ is sampled.

This means that if we start at v and follow suffix links successively, *i.e.*, v, SLINK(v), SLINK(SLINK(v)), ..., we will find a sampled node in at most δ steps. Note that this property implies that the ROOT must be sampled, since SLINK(ROOT) is undefined. We now show that it is possible to δ-sample a suffix tree.

THEOREM 4.2. *There exists a δ-sampled tree S for any suffix tree $\mathcal{T}$.*

PROOF. We sample the nodes v such that $\text{SDEP}(v) \equiv_{\delta/2} 0$ and there is another node v' such that $v = \text{SLINK}^{\delta/2}(v')$. Since $\text{SDEP}(\text{SLINK}^i(v')) = \text{SDEP}(v') - i$, this guarantees that, for any v' such that $\text{SDEP}(v') \geq \delta/2$, the $\text{SDEP}(\text{SLINK}^i(v')) \equiv_{\delta/2} 0$ condition holds for exactly two values in the range $i \in [0, \delta - 1]$. For the largest of those two i values, the second sampling condition must hold as well. (If $\text{SDEP}(v') < \delta/2$, v' is sufficiently close to the ROOT, which is sampled.) On the other hand, for each sampled node $v \neq \text{ROOT}$, there are at least other $\delta/2 - 1$ non-sampled nodes that point to it via SLINK^i, as their SDEP is not a multiple of $\delta/2$. Hence there are $s \leq 1 + t/(2\delta) = O(t/\delta) = O(n/\delta)$ sampled nodes. □

We represent the sampled tree S as a sequence of balanced parentheses, using Theorem 2.5. Operations PREORDER_S, PREORDER_S^{-1}, PARENT_S, TLAQ_S, LCA_S, and TDEP_S, are all supported in constant time and $O(n/\delta)$ bits of space. We will also need to store, in arrays indexed by $\text{PREORDER}_S(v)$, the values $\text{SDEP}(v)$ and $\text{TDEP}(v)$ for the sampled nodes (do not confuse $\text{TDEP}(v) = \text{TDEP}_{\mathcal{T}}(v)$, the depth of a sampled node v in the suffix tree, with $\text{TDEP}_S(v)$, the depth of v in the sampled tree). These arrays require $O((n/\delta)\log n)$ bits of space.

In the dynamic case we use Theorem 2.6 to represent S with balanced parentheses. This takes $O(n/\delta)$ bits and supports operations PREORDER_S, PREORDER_S^{-1}, PARENT_S, LCA_S, and TDEP_S, all in $O(\log n/\log\log n)$ time. The structure also supports insertion and deletion of leaves and unary nodes. The representation also needs to maintain the SDEP values of nodes in S, which are handled using a simple dynamic structure such as that presented by Navarro and Sadakane [2010]: It allows inserting, deleting and accessing the values in $O(\log n/\log\log n)$ time while using $O((n/\delta)\log n)$ bits of space.

In order to make effective use of the sampled tree, we need a way to map any node v to its *lowest sampled ancestor*, LSA(v). Another important operation is the *lowest common sampled ancestor* $\text{LCSA}(v, v') = \text{LSA}(\text{LCA}(v, v'))$, *i.e.*, lowest common ancestor in the sampled tree S. In our example, LCSA$(3, 4)$ is the ROOT, whereas LCA$(3, 4)$ is $[3, 6]$, *i.e.*, the node labeled b. The next lemma shows how the general LCSA and LSA queries can be answered if LSA *for leaves* is available, and then we go on to solve that specific problem. The mapping will also let us compute the range $[v_l, v_r]$ of a sampled node v.

LEMMA 4.3. *Let v and v' be nodes of a suffix tree $\mathcal{T}$ and S an δ-sampled subtree, then the following properties always hold:*

$$\begin{aligned}
v_1 \textit{ ancestor of } v_2 \;&\Rightarrow\; \text{LSA}(v_1) \textit{ ancestor of } \text{LSA}(v_2) && (1)\\
\text{LCSA}(v, v') &= \text{LCA}_S(v, v')\textit{, when } v \textit{ and } v' \textit{ belong to } S && (2)\\
\text{LCSA}(v, v') &= \text{LCSA}(\text{LSA}(v), \text{LSA}(v')) && (3)\\
\text{LCSA}(v, v') &= \text{LCA}_S(\text{LSA}(v), \text{LSA}(v')) && (4)\\
\text{LSA}(v) &= \text{LCA}_S(\text{LSA}(v_l), \text{LSA}(v_r)) && (5)
\end{aligned}$$

PROOF. For (1), LSA(v_1) is transitively an ancestor of v_2 and it is sampled, thus by definition of LSA it is also an ancestor of LSA(v_2).

For the rest of the proof let us define $v'' = \text{LCSA}(v, v') = \text{LSA}(\text{LCA}(v, v'))$. For Eq. (2) note that v'' is a node of S and it is an ancestor of both v and v', since it is an ancestor of $\mathrm{LCA}(v, v')$. Therefore by the definition of LCA_S we conclude that v'' is an ancestor of $v''' = \mathrm{LCA}_S(v, v')$. On the other hand v''' is an ancestor of v and of v'. Therefore v''' is an ancestor of $\mathrm{LCA}(v, v')$. Taking LSA on both nodes (1) we have that $\mathrm{LSA}(v''') = v'''$ is an ancestor of v''.

For Eq. (3) note that since $\mathrm{LCA}(v, v')$ is an ancestor of v, we have by (1) that v'' is an ancestor of $\mathrm{LSA}(v)$, and likewise of $\mathrm{LSA}(v')$. Hence v'' is an ancestor of $\mathrm{LCA}(\mathrm{LSA}(v), \mathrm{LSA}(v'))$. Therefore by (1) $\mathrm{LSA}(v'') = v''$ is an ancestor of $v^* = \mathrm{LSA}(\mathrm{LCA}(\mathrm{LSA}(v), \mathrm{LSA}(v'))) = \mathrm{LCSA}(\mathrm{LSA}(v), \mathrm{LSA}(v'))$. On the other hand $\mathrm{LSA}(v)$ is an ancestor of v, likewise $\mathrm{LSA}(v')$ is an ancestor of v'. Therefore $\mathrm{LCA}(\mathrm{LSA}(v), \mathrm{LSA}(v'))$ is an ancestor of $\mathrm{LCA}(v, v')$. Hence v^* is an ancestor of v''.

For Eq. (4) observe that $\mathrm{LSA}(v)$ and $\mathrm{LSA}(v')$ belong to S, hence we can use Eq. (2) to conclude that $\mathrm{LCSA}(\mathrm{LSA}(v), \mathrm{LSA}(v')) = \mathrm{LCA}_S(\mathrm{LSA}(v), \mathrm{LSA}(v'))$. Using Eq. (3) we can replace the first term by $\mathrm{LCSA}(v, v')$.

For Eq. (5) note that $v = \mathrm{LCA}(v_l, v_r)$. Therefore using Eq. (4) we have that $\mathrm{LSA}(v) = \mathrm{LSA}(\mathrm{LCA}(v_l, v_r)) = \mathrm{LCSA}(v_l, v_r) = \mathrm{LCA}_S(\mathrm{LSA}(v_l), \mathrm{LSA}(v_r)))$. □

4.1. Computing LSA for Leaves

We use the following data structures to provide the mapping LSA between leaves $v = [v, v]$ and their lowest sampled ancestors in S, and conversely, to obtain the range $[v_l, v_r]$ for sampled nodes v.

(1) We will identify S with its balanced parentheses representation $S[0, 2s - 1]$, so that we will speak indistinctly of nodes in S and their tree operations, and positions in S and their parenthesis operations.

(2) A bitmap $B[n + 2s]$ containing $2s$ ones, which correspond to the parentheses of S, and n zeros, which correspond to the suffix tree leaves. If leaf v is contained in the sampled node represented by parentheses $S[u]$ and $S[u']$, then the 0 bit corresponding to v must be placed between the $(u+1)$-th and the $(u'+1)$-th 1 of B. Since B contains $2s$ ones, its representation using Theorem 2.3 requires at most $2s \log \frac{n+2s}{2s} + O(s) + o(n + 2s) = O((n/\delta) \log \delta)$ bits of space, and supports constant-time $\textsc{Rank}$ and $\textsc{Select}$.

In our example $S = (()())$ and $B = 1000101101001$, see Figure 3. An operational way to describe B, which is useful to explain later the dynamic case, is as follows: Initialize it with n bits all equal to 0. Now, for every sampled node $v = [v_l, v_r]$, insert a 1 at $\textsc{Select}_0(B, v_l)$ and another right after $\textsc{Select}_0(B, v_r)$.

To compute LSA we use an auxiliary function defined as follows:

$$\textsc{Pred}(v) \;=\; \textsc{Rank}_1(B, \textsc{Select}_0(B, v)) - 1 \;=\; \textsc{Select}_0(B, v) - v - 1,$$

which gives the position of the last parenthesis in S preceding leaf v.

LEMMA 4.4. *Let $p = \textsc{Pred}(v)$. If $S[p] =$ '(' then p is the lowest sampled ancestor of v; otherwise it is $\textsc{Enclose}_S(\textsc{FindMatch}_S(p))$.*

PROOF. If $S[p] =$ '(', then $\textsc{FindMatch}_S(p)$ closes after leaf v, and by definition of B, v is contained in p. The descendants of p, if any, start after v and hence do not contain it. If $S[p] =$ ')' then, by definition of B, $p' = \textsc{FindMatch}_S(p)$ does not contain v (it closes before v) nor does its next sibling, if any (it opens after v), but $\textsc{Enclose}_S(p')$ opens before and closes after v and hence it is the lowest node containing v. □

Now we are ready to give the formula for $\mathrm{LSA}(v)$. Notice that $v = [v, v]$ is a leaf position, and the answer is an identifier in tree S.

$$\begin{aligned} p &= \textsc{Pred}(v) \\ \mathrm{LSA}(v) &= \textbf{if } S[p] = \texttt{'('} \textbf{ then } p \textbf{ else } \textsc{Enclose}_S(\textsc{FindMatch}_S(p)) \end{aligned}$$

Consider for example the leaf numbered 5 in Figure 3. This leaf is not sampled, but in the original tree it appears between leaf 4 and the end of the tree, more specifically between parenthesis ')' of 4 and parenthesis ')' of the ROOT. Thus $\text{PRED}(5) = 4$. In this case, since the parenthesis we obtain is a ')', we know that LSA is the parent of that node.

In the opposite direction, we wish to find out the leaf interval $[v_l, v_r]$ corresponding to a sampled node identifier v of S. This is not hard to do:

$$\begin{aligned} v_l &= \text{RANK}_0(\text{SELECT}_1(B, v)) \\ v_r &= \text{RANK}_0(\text{SELECT}_1(B, \text{FINDMATCH}_S(v))) - 1 \end{aligned}$$

Summarizing, we can map from sampled nodes in S to suffix tree nodes $[v_l, v_r]$, as well as the reverse with operations LSA and LCSA, all in constant time and using $O((n/\delta)\log\delta)$ bits of space.

In the dynamic case, we use Theorem 2.4 to handle B and Theorem 2.6 to handle S. This retains the same space, and operations cost $O(\log n/\log\log n)$ time. The update operations we will need to carry out are: (i) insertion/deletion of leaves in B, when a leaf appears in / disappears from the suffix tree $\mathcal{T}$, and (ii) insertion/deletion of pairs of matching parentheses in/from S (and their corresponding 1s in B), when nodes become sampled/unsampled in S. Figure 3 illustrates the effect of sampling $b = [3, 6]$ in our running example.

For (i), we will want to insert a new leaf (that is, suffix array position) between leaves $v - 1$ and v. If $v - 1$ and v are consecutive in B, *i.e.*, $\text{SELECT}_0(B, v-1) + 1 = \text{SELECT}_0(B, v)$, then we simply do $\text{INSERT}(B, \text{SELECT}_0(B, v), 0)$. Yet in general there could be several sampled nodes containing the leaf. Thus the general procedure is as follows. The new leaf is a child of some internal node v' of $\mathcal{T}$. We assume that in case v' had to be sampled due to the update, it is already in S. Before the new leaf is inserted in B, since v' cannot be unary, it is an ancestor of leaves $v - 1$ or v or both. Let us assume v' is ancestor of $v - 1$; the other case is similar. We compute $t = \text{TDEP}_S(\text{LSA}(v-1)) - \text{TDEP}_S(\text{LSA}(v'))$ and run $\text{INSERT}(B, \text{SELECT}_0(B, v-1)+t+1, 0)$. To remove leaf number v we run $\text{DELETE}(B, \text{SELECT}_0(B, v))$.

For (ii), the insertion of a new node $v = [v_l, v_r]$ in the sampled tree translates into the insertion of a matching parentheses pair at positions (u, u') in S. For example if the new node encloses current node v then $u = v$ and $u' = \text{FINDMATCH}_S(v)$; if it is a leaf first child of v then $u = u' = v + 1$; if it is a leaf next sibling of v then $u = u' = \text{FINDMATCH}_S(v)+1$. After carrying out the insertion on S (via $\text{INSERT}_S(u, u'+1)$), we must update B. We compute $m' = \max(\text{SELECT}_1(B, u'), \text{SELECT}_0(B, v_r))$ for $\text{INSERT}(B, m' + 1, 1)$ and then $m = \max(\text{SELECT}_1(B, u), \text{SELECT}_0(B, v_l))$ for $\text{INSERT}(B, m, 1)$. For removing sampled node v, after $\text{DELETE}_S(u, u')$ for $u = v$ and $u = \text{FINDMATCH}_S(v)$, we also update B by $\text{DELETE}(B, \text{SELECT}_1(B, u'))$ and then $\text{DELETE}(B, \text{SELECT}_1(u))$.

Thus, all the updates required for the dynamic case can be carried out in $O(\log n/\log\log n)$ time per update to S or to $\mathcal{T}$.

5. SUFFIX TREE NAVIGATION

We start this section by showing in Lemma 5.1 a simple relation between the SLINK and LCA operations, and use this relation to obtain an algorithmic way of computing the SDEP value of non-sampled nodes[8], in Lemma 5.2. This algorithmic procedure

[8]A detailed exposition on why these properties are important for representing suffix trees is given in Appendix 9.

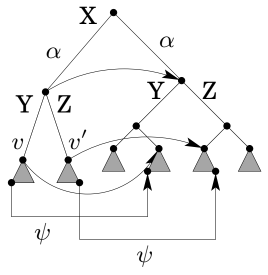

Fig. 4. Schematic representation of the relation between LCA and SLINK, see Lemma 5.1. Curved arrows represent SLINK and straight arrows the ψ function.

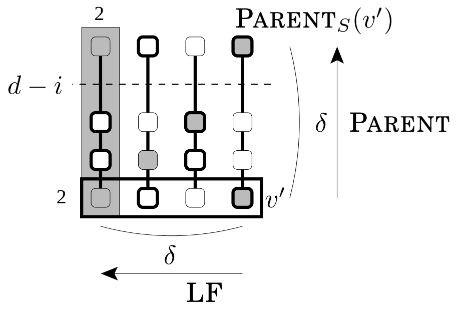

Fig. 5. Schematic representation of the $v_{i,j}$ nodes of the SLAQ operation. The nodes sampled because of Definition 4.1 are in bold and those sampled because of the condition of TDEP are filled.

turns out to be flexible enough to support a complete spectrum of operations, which we explain throughout this section.

LEMMA 5.1. *For any nodes v, v' such that* $\text{LCA}(v, v') \neq$ ROOT *we have that* $\text{SLINK}(\text{LCA}(v, v')) = \text{LCA}(\text{SLINK}(v), \text{SLINK}(v'))$.

PROOF. Assume that the path-labels of v and v' are respectively $X.\alpha.Y.\beta$ and $X.\alpha.Z.\beta'$, where $Y \neq Z$. According to the definitions of LCA and SLINK, we have that $\text{LCA}(v, v') = X.\alpha$ and $\text{SLINK}(\text{LCA}(v, v')) = \alpha$. On the other hand the path-labels of $\text{SLINK}(v)$ and $\text{SLINK}(v')$ are respectively $\alpha.Y.\beta$ and $\alpha.Z.\beta'$. Therefore the path-label of $\text{LCA}(\text{SLINK}(v), \text{SLINK}(v'))$ is also α. Hence this node must be the same as $\text{SLINK}(\text{LCA}(v, v'))$. ☐

Figure 4 illustrates this lemma; ignore the nodes associated with ψ. The condition $\text{LCA}(v, v') \neq$ ROOT is easy to verify, in a suffix tree, by comparing the first letters of the path-label of v and v', *i.e.* $\text{LCA}(v, v') \neq$ ROOT iff $v[0] = v'[0]$.

The next lemma shows a fundamental property that relates the kernel operations LCA, SLINK, and SDEP.

LEMMA 5.2. *Let v, v' be nodes in a δ-sampled suffix tree such that* $\text{SLINK}^r(\text{LCA}(v, v')) =$ ROOT, *and let* $d = \min(\delta, r+1)$. *Then*

$$\text{SDEP}(\text{LCA}(v, v')) = \max_{0 \leq i < d}\{i + \text{SDEP}(\text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v')))\}$$

PROOF. The following reasoning holds for any valid i:

$$\begin{aligned} \text{SDEP}(\text{LCA}(v, v')) &= i + \text{SDEP}(\text{SLINK}^i(\text{LCA}(v, v'))) && (6)\\ &= i + \text{SDEP}(\text{LCA}(\text{SLINK}^i(v), \text{SLINK}^i(v'))) && (7)\\ &\geq i + \text{SDEP}(\text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v'))) && (8) \end{aligned}$$

Eq. (6) holds by iterating the fact that $\text{SDEP}(v'') = 1 + \text{SDEP}(\text{SLINK}(v''))$ for any node v'' for which $\text{SLINK}(v'')$ is defined. Eq. (7) results from applying Lemma 5.1 repeatedly. Inequality (8) comes from the definition of LCSA and the fact that if node v''' is an ancestor of node v'' then $\text{SDEP}(v'') \geq \text{SDEP}(v''')$. Therefore $\text{SDEP}(\text{LCA}(v, v')) \geq \max_{0 \leq i < d}\{\ldots\}$. On the other hand, from Definition 4.1 we know that for some $i < \delta$ the node $\text{SLINK}^i(\text{LCA}(v, v'))$ is sampled. The formula goes only up to d, but $d < \delta$ only if $\text{SLINK}^d(\text{LCA}(v, v')) =$ ROOT, which is also sampled. According to the definition of LCSA, inequality (8) becomes an equality for that node. Hence $\text{SDEP}(\text{LCA}(v, v')) \leq \max_{0 \leq i < d}\{\ldots\}$. ☐

To apply Lemma 5.2 we need to support operations LCSA, SDEP, and SLINK. Operation LCSA is supported in constant time (Section 4). Since SDEP is applied only to sampled nodes, we have it readily stored in the sampled tree. Hence the only obstacle is SLINK. Sadakane [2007] showed that $\text{SLINK}(v) = \text{LCA}(\psi(v_l), \psi(v_r))$, whenever $v \neq \text{ROOT}$. This is, now, a trivial consequence of Lemma 5.1 since $\text{SLINK}(v) = \text{SLINK}(\text{LCA}(v_l, v_r)) = \text{LCA}(\text{SLINK}(v_l), \text{SLINK}(v_r))) = \text{LCA}(\psi(v_l), \psi(v_r))$. This is not necessarily equal to the interval $[\psi(v_l), \psi(v_r)]$, see node $X.\alpha$ in Figure 4, or consider the example of node bb represented by the interval $[5, 6]$ for which $\text{SLINK}(v) = [3, 6]$ and $[\psi(5), \psi(6)] = [4, 5]$. In general $\text{SLINK}^i(v) = \text{LCA}(\psi^i(v_l), \psi^i(v_r))$, which can be shown by induction using Lemma 5.1: $\text{SLINK}^{i+1}(v) = \text{SLINK}(\text{SLINK}^i(v)) = \text{SLINK}(\text{LCA}(\psi^i(v_l), \psi^i(v_r))) = \text{LCA}(\text{SLINK}(\psi^i(v_l)), \text{SLINK}(\psi^i(v_r))) = \text{LCA}(\psi^{i+1}(v_l), \psi^{i+1}(v_r))$.

Now we have the tools to support LCA using Lemma 5.2.

LEMMA 5.3. $\text{LCA}(v, v') = \text{LF}(v[0..i-1], \text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v')))$ *for any nodes* v, v'*, where* i *is the value maximizing the formula of Lemma 5.2.*

PROOF. This is direct from Lemma 5.2. Let i be such that $\text{SLINK}^i(\text{LCA}(v, v'))$ is sampled. From Lemma 5.1 $\text{SLINK}^i(\text{LCA}(v, v')) = \text{LCA}(\text{SLINK}^i(v), \text{SLINK}^i(v'))$, which is also the same as $\text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v'))$ because it is a sampled node. Note that for the LF mapping we have that $\text{LF}(v''[0], \text{SLINK}(v'')) = v''$. Applying this iteratively to $\text{SLINK}^i(\text{LCA}(v, v'))$ we obtain the equality in the lemma. □

To use this lemma we must know which is the correct i. This is easily determined if we first compute $\text{SDEP}(\text{LCA}(v, v'))$. Accessing the letters to apply LF is not a problem, as it suffices to obtain the first letter of a path label, $\text{SLINK}^i(v)[0] = \text{SLINK}^i(v')[0]$. But we are stuck in a circular dependency between LCA and SLINK.

5.1. Solving the Kernel Operations

To get out of this dependency we will handle all the computation over leaves, for which we can compute $\text{SLINK}(v) = \psi(v)$ using the CSA.

LEMMA 5.4. *For any two suffix tree nodes* v, v' *we have* $\text{LCA}(v, v') = \text{LCA}(\min\{v_l, v'_l\}, \max\{v_r, v'_r\})$.

PROOF. Let v'' and v''' be respectively the nodes on the left and on the right of the equality. Assume that they are represented as $[v''_l, v''_r]$ and $[v'''_l, v'''_r]$ respectively. Hence $v''_l \leq v_l, v'_l$ and $v''_r \geq v_r, v'_r$ since v'' is an ancestor of v and v'. This means that $v''_l \leq \min\{v_l, v'_l\} \leq \max\{v_r, v'_r\} \leq v''_r$, *i.e.*, v'' is also an ancestor of $\min\{v_l, v'_l\}$ and $\max\{v_r, v'_r\}$. Since v''' is by definition the lowest common ancestor of these nodes we have that $v''_l \leq v'''_l \leq v'''_r \leq v''_r$. Using a similar reasoning for v''' we conclude that $v'''_l \leq v''_l \leq v''_r \leq v'''_r$ and hence $v'' = v'''$. □

Observe this property in Figure 4; ignore SLINK, ψ and the subtree on the right. Using this property and ψ the equation in Lemma 5.2 reduces to

$$\begin{aligned}
\text{SDEP}(\text{LCA}(v, v')) &= \text{SDEP}(\text{LCA}(\min\{v_l, v'_l\}, \max\{v_r, v'_r\})) \\
&= \max_{0 \leq i < d}\{i + \text{SDEP}(\text{LCSA}(\text{SLINK}^i(\min\{v_l, v'_l\}), \text{SLINK}^i(\max\{v_r, v'_r\}))))\} \\
&= \max_{0 \leq i < d}\{i + \text{SDEP}(\text{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\}))))\}
\end{aligned}$$

Operationally, this corresponds to iteratively taking the ψ function, δ times or until the ROOT is reached. At each step we find the LCSA of the two current leaves and retrieve its stored SDEP. The overall process takes $O(t_\psi \delta)$ time. Note that in the dynamic scenario the rank and tree operations are slower by an $O(\frac{\log n}{\log \log n})$ factor. Likewise SDEP and LCA simplifies to

$$\begin{aligned}\text{SDEP}(v) &= \text{SDEP}(\text{LCA}(v,v)) = \max_{0 \le i < d}\{i + \text{SDEP}(\text{LCSA}(\psi^i(v_l), \psi^i(v_r)))\}\\ \text{LCA}(v,v') &= \text{LF}(v[0..i-1], \text{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\})))\end{aligned}$$

Now it is finally clear that we do not need SLINK to compute LCA. The time to compute LCA is thus $O((t_\psi + t_{\mathsf{LF}})\delta)$, and that to compute SDEP is $O(t_\psi \delta)$. Using LCA we compute $\text{SLINK}(v) = \text{LCA}(\psi(v_l), \psi(v_r))$ in $O((t_\psi + t_{\mathsf{LF}})\delta)$ and $\text{SLINK}^i = \text{LCA}(\psi^i(v_l), \psi^i(v_r))$ in $O(t_{\mathsf{SA}} + (t_\psi + t_{\mathsf{LF}})\delta)$ time. Note that the arguments to LCSA do not correspond necessarily to nodes, but the formulas hold in this case too.

THEOREM 5.5. *Suffix tree operations* SDEP, LCA, SLINK, *and* SLINK^i *can be computed respectively in* $O(t_\psi \delta)$, $O((t_\psi + t_{\mathsf{LF}})\delta)$, $O((t_\psi + t_{\mathsf{LF}})\delta)$, *and* $O(t_{\mathsf{SA}} + (t_\psi + t_{\mathsf{LF}})\delta)$ *time, provided a CSA implements* ψ *in* $O(t_\psi)$, LF *in* $O(t_{\mathsf{LF}})$, *and* A *and* A^{-1} *in* $O(t_{\mathsf{SA}})$ *time, and we have a* δ*-sampled suffix tree.*

5.2. Further Operations

We now show how other operations can be computed on top of the kernel ones.

Computing PARENT(v). For any node v represented as $[v_l, v_r]$ we have that PARENT(v) is either $\text{LCA}(v_l - 1, v_l)$ or $\text{LCA}(v_r, v_r + 1)$, whichever is lowest. This is because suffix trees are compact and hence PARENT$([v_l, v_r])$ must contain $[v_l - 1, v_r]$ or $[v_l, v_r + 1]$. Notice that if one of these nodes is undefined, either because $v_l = 0$ or $v_r = n - 1$, then the parent is the other node. If both nodes are undefined then node v is the ROOT, which has no PARENT. The time is $O((t_\psi + t_{\mathsf{LF}})\delta)$.

Computing CHILD(v, X). We show how CHILD can be computed in a general and efficient way directly over the CSA. The *generalized branching* for nodes v_1 and v_2 consists in determining the node with path label $v_1.v_2$ if it exists. A simple solution is to binary search the interval of v_1 for the subinterval of the v''s such that $\psi^m(v') \in v_2$, where $m = \text{SDEP}(v_1)$. This approach requires $O(t_{\mathsf{SA}} \log n)$ time and it was first considered using CSAs by Huynh et al. [2006]. Thus we are able to compute CHILD(v, X), using v_2 as the subinterval of A where the suffixes start with X. This is easily computed from the CSA as WEINERLINK(X, ROOT).

This general solution can be improved by noticing that we are using SLINK^m at arbitrary positions of the CSA for the binary search. Recall that SLINK^m is solved via A and A^{-1}, *i.e.*, $\text{SLINK}^m(v_l) = \psi^m(v_l) = A^{-1}[A[v_l]+m]$. Thus, we could sample A and A^{-1} regularly so as to store their values explicitly. That is, we explicitly store the values $A[j\theta]$ and $A^{-1}[j\theta]$ for all j.[9] To solve a generalized branching, we start by building a table of ranges $D[0] = v_2$ and $D[i] = \text{LF}(v_1[m-i..m-1], v_2)$, for $1 \le i < \theta$. If $m < \theta$ the answer is $D[m]$. Otherwise, we binary search the interval of v_1, accessing only the sampled elements of A. To determine the branching we should compute $\psi^m(j\theta) = A^{-1}[A[j\theta]+m]$ for some $j\theta$ values in v_1. To use the cheaper sampled A^{-1} as well, we need that $A[j\theta]+m$ be divisible by θ, thus we instead compute $\psi^{m'}$ for $m' = \lfloor (A[j\theta]+m)/\theta \rfloor \theta - A[j\theta]$. Hence instead of verifying that $\psi^m(j\theta) \in v_2$, we verify that $\psi^{m'} \in D[m - m']$. After this process we still have to binary search an interval of size $O(\theta)$, which is carried out naively.

[9]Most CSAs already include such a sampling in one way or another [Navarro and Mäkinen 2007].

The overall process requires time $O(t_{\mathsf{SA}} + (t_\psi + t_{\mathsf{LF}})\theta)$ to access the last letters of v_1 and build D, plus $O(\log(n/\theta))$ for binary searching the samples; plus $O(t_{\mathsf{SA}} \log \theta)$ for the final binary search. For example we can just use $\theta = \delta$.[10] In this case the time is $O((t_\psi + t_{\mathsf{LF}})\delta + t_{\mathsf{SA}} \log \delta + \log n)$ and the extra space is $O((n/\delta)\log n)$ bits. This is the value used in Tables V and VI.

Yet, regarding our discussion in Section 8, we wish to avoid more extra spaces of this magnitude. For the particular case of using the FM-index (under which we get our best results), we can do the following to achieve the same time with less extra space. Set $\theta = \frac{t_{\mathsf{SA}} \log t_{\mathsf{SA}}}{t_{\mathsf{LF}}}$, so that the overall time is $O(\log n + (1 + t_\psi/t_{\mathsf{LF}})t_{\mathsf{SA}} \log t_{\mathsf{SA}})$ (which is $O(\log n (\log\log n)^2)$ in Table I), and the extra space for the sampling is $O((n/\theta)\log n)$. Recall Table III, where we defined $t_{\mathsf{SA}} = l\, t_{\mathsf{LF}}$ and chose $l = \log n \log\log n$, to have $O(\frac{n}{\log\log n}) = o(n)$ extra bits of space for the CSA. Hence $O((n/\theta)\log n) = O(\frac{n \log n}{l \log l})$. This is less than the $O((n/l)\log n)$ bits paid by the CSA for its own sampling. For the value of l we have chosen, it is $O(\frac{n}{(\log\log n)^2})$.

In a dynamic scenario, we do not store exactly the $A[j\theta]$ values; instead we guarantee that for any k there is a k' such that $k-\theta < k' \leq k$ and $A[k']$ is sampled, and the same for A^{-1}. Still the sampled elements of A and the m' to use can be easily obtained in $O(\log n)$ time. Those sampled sequences are not hard to maintain upon insertions/deletions in A. For example, Mäkinen and Navarro [2008, Sec. 7.1] describe how to maintain A^{-1} (called S_C in there), and essentially how to maintain A (called S_A in there; the only missing point is how to maintain approximately spaced samples in A, which can be done exactly as for A^{-1}). Thus the space remains the same and the $O(\log n)$ term in the complexity becomes $O(\log^2 n)$.

Computing $\textrm{TDEP}(v)$. To compute TDEP we add other $O(n/\delta)$ nodes to the sampled tree S so as to guarantee that, for any suffix tree node v, $\textrm{PARENT}^j(v)$ is sampled for some $0 \leq j < \delta$. Recall that the $\textrm{TDEP}(v)$ values are stored in S. Since $\textrm{TDEP}(v) = \textrm{TDEP}(\textrm{LSA}(v))+j$, where $\textrm{LSA}(v) = \textrm{PARENT}^j(v)$, $\textrm{TDEP}(v)$ can be computed by reading $\textrm{TDEP}(\textrm{LSA}(v))$ and adding the number of nodes between v and $\textrm{LSA}(v)$. The sampling guarantees that $j < \delta$. Hence to determine j we iterate PARENT until reaching $\textrm{LSA}(v)$. The total cost is $O((t_\psi + t_{\mathsf{LF}})\delta^2)$.

To achieve this sampling property, we sample the nodes v such that $\textrm{TDEP}(v) \equiv_{\delta/2} 0$ and $\textrm{HEIGHT}(v) \geq \delta/2$. Since $\textrm{TDEP}(\textrm{PARENT}^i(v)) = \textrm{TDEP}(v) - i$, the first condition holds for exactly two i's in $[0, \delta-1]$ if $\textrm{TDEP}(v) \geq \delta/2$. Since HEIGHT is strictly increasing, the second condition holds for sure for the largest i. On the other hand, since every sampled node has at least $\delta/2$ descendants that are not sampled, it follows that we sample $O(n/\delta)$ extra nodes with this criterion.

We are unable to maintain either the sampled TDEP values or the sampling property in the dynamic scenario. Therefore, this operation and next two are not supported in the dynamic case.

Computing $\textrm{TLAQ}(v, d)$. We extend the notation $\textrm{PARENT}_S(v)$ to represent $\textrm{LSA}(v)$ when v is not sampled. Recall that the sampled tree supports constant-time level ancestor queries. Hence we have any $\textrm{PARENT}_S^i(v)$ in constant time for any node v and any i. We binary search $\textrm{PARENT}_S^i(v)$ to find the sampled node v' with $\textrm{TDEP}(v') \geq d > \textrm{TDEP}(\textrm{PARENT}_S(v'))$. Notice that this can be computed by evaluating only the second inequality, which refers to sampled nodes only. Now we iterate the PARENT operation, from v', exactly $\textrm{TDEP}(v') - d$ times. We need the additional sampling introduced for TDEP to guarantee $\textrm{TDEP}(v') - d < \delta$. Hence the total time is $O(\log n + (t_\psi + t_{\mathsf{LF}})\delta^2)$.

[10]This speedup immediately improves the results of Huynh et al. [2006].

Computing $\text{SLAQ}(v, d)$. We start by binary searching for the value m such that $v' = \text{PARENT}_S^m(\text{SLINK}^{\delta-1}(v))$ satisfies $\text{SDEP}(v') \geq d - (\delta - 1) > \text{SDEP}(\text{PARENT}_S(v'))$. Now we scan all the sampled nodes $v_{i,j} = \text{PARENT}_S^j(\text{LSA}(\text{LF}(v[i..\delta - 1], v')))$ with $\text{SDEP}(v_{i,j}) \geq d - i$ and $i, j < \delta$. This means that we start at node v', follow LF, reduce every node found to the sampled tree S and use PARENT_S until the SDEP of the node drops below $d - i$. Our aim is to find the $v_{i,j}$ that minimizes $\text{SDEP}(v_{i,j}) - (d - i) \geq 0$, and then apply the LF mapping to it.

The time to perform this operation depends on the number of existing $v_{i,j}$ nodes. For this operation the sampling must satisfy Definition 4.1 and the condition for computing TDEP. Each condition contributes with at most two sampled nodes for every δ nodes in one direction (SLINK or PARENT). Therefore, there are at most 4δ nodes $v_{i,j}$ (see Figure 5), and thus the time is $O(\log n + (t_\psi + t_{\mathsf{LF}})\delta)$. Unfortunately, the same trick does not work for TDEP and TLAQ, because we cannot know which is the “right” node without bringing all of them back with LF.

Computing FCHILD. To find the first child of $v = [v_l, v_r]$, where $v_l \neq v_r$, we simply compute $\text{SLAQ}(v_l, \text{SDEP}(v) + 1)$. Likewise if we use v_r we obtain the last child. It is possible to avoid the binary search step of SLAQ by choosing $v' = \text{PARENT}_S^m(\text{LSA}(\text{SLINK}^{\delta-1}(v_l)))$ for $m = \text{TDEP}_S(\text{LSA}(\text{SLINK}^{\delta-1}(v_l))) - \text{TDEP}_S(\text{LSA}(\text{SLINK}^{\delta-1}(v))) - 1$, if $i \geq 0$, and $v' = \text{SLINK}^{\delta-1}(v_l)$ if $m = -1$. Thus the time is $O((t_\psi + t_{\mathsf{LF}})\delta)$.

In the dynamic case we do not have SLAQ. Instead, $\text{FCHILD}(v)$ can be determined by computing $X = \text{LETTER}(v_l, \text{SDEP}(v)+1)$ and then $\text{CHILD}(v, X)$. The time for CHILD dominates.

Computing NSIB. The next sibling of $v = [v_l, v_r]$ can be computed as $\text{SLAQ}(v_r + 1, \text{SDEP}(\text{PARENT}(v)) + 1)$ for any $v \neq \text{ROOT}$. Likewise we can obtain the previous sibling with $v_l - 1$. We must check that the answer has the same parent as v, to cover the case where there is no previous/next sibling. We can also skip the binary search.

Again, in the dynamic case $\text{NSIB}(v)$ can be computed with CHILD. If $\text{PARENT}(v) = [v'_l, v'_r]$ and $v'_r > v_r$, then we compute $X = \text{LETTER}(v_r + 1, \text{SDEP}(v) + 1)$ and do $\text{CHILD}(v', X)$.

We are ready to state our summarizing theorem.

THEOREM 5.6. *Using a compressed suffix array (CSA) with the properties stated in Table III, it is possible to represent a suffix tree with the properties given in Table V (FCST).*

Table V also compares our FCST with the CST [Sadakane 2007] and the EBST [Fischer et al. 2009; Fischer 2010]. The times for the EBST are slightly simplified as they depend on other parameters. The best for the FCST is to use the FM-index [Ferragina et al. 2007], which reaches the minimum space, whereas FCST times do not improve by using other CSAs because its operations depend on $t_\psi + t_{\mathsf{LF}}$. The alternatives, instead, depend mostly on t_{SA} and t_ψ, so they improve significantly by using a slightly larger CSA [Grossi et al. 2003] that offers much better times for t_ψ and t_{SA} but slower t_{LF}, see Table III.

The operations in Table V provide an extremely functional suffix tree. Yet, not all the potentially interesting operations are supported. A notorious deficiency is the inability to efficiently compute the $\text{PREORDER}_{\mathcal{T}}$ value of a suffix tree node. This is essential when we need to associate satellite information to nodes.

We propose an alternative scheme for this problem. The technique applies only for internal nodes and not leaves, which can be indexed separately by their position in the

Table V. Comparison between compressed suffix tree representations. We omit the operations that are carried out directly on the CSA, see Table III. We simplify the FCST complexities by assuming $\delta = \omega(\log n)$ as otherwise the extra space is not $o(n)$. We also assume that $t_\psi, t_{\mathsf{LF}}, t_{\mathsf{SA}} = \Omega(t_{\mathsf{tree}})$. The f of EBST must be $O(\frac{\log n}{\log\log n})$ and $\Omega(\log^{[r]} n)$ for some constant $r \geq 0$, which denotes r applications of $\log$ to n. For EBST "not supported" means that it needs at least twice the space to support those operations. Notice that CHILD can, alternatively, be computed using FCHILD and at most σ times NSIB.

	CST	EBST	Ours (FCST)
Space in bits	$\|CSA\| + \mathbf{6n} + o(n)$	$\|CSA\| + O(n/f)$	$\|CSA\| + O((n/\delta)\log n)$
ROOT	$O(1)$	$O(1)$	$O(1)$
COUNT	$O(1)$	$O(1)$	$O(1)$
ANCESTOR	$O(1)$	$O(1)$	$O(1)$
PARENT	t_{tree}	$t_{\mathsf{SA}} f \log\log n$	$(t_\psi + t_{\mathsf{LF}})\delta$
FCHILD	t_{tree}	$t_{\mathsf{SA}} f \log^2 f$	$(t_\psi + t_{\mathsf{LF}})\delta$
NSIB	t_{tree}	$t_{\mathsf{SA}} f(\log^2 f + \log\log n)$	$(t_\psi + t_{\mathsf{LF}})\delta$
LCA	t_{tree}	$t_{\mathsf{SA}} f(\log^2 f + \log\log n)$	$(t_\psi + t_{\mathsf{LF}})\delta$
TDEP	t_{tree}	*Not supported*	$(t_\psi + t_{\mathsf{LF}})\delta^2$
TLAQ	t_{tree}	*Not supported*	$(t_\psi + t_{\mathsf{LF}})\delta^2$
CHILD	$t_{\mathsf{SA}} \log\sigma$	$t_{\mathsf{SA}}(f\log^2 f + \log\sigma)$	$t_{\mathsf{SA}} \log\delta + (t_\psi + t_{\mathsf{LF}})\delta$
SLINK	t_ψ	$t_{\mathsf{SA}} f(\log^2 f + \log\log n)$	$(t_\psi + t_{\mathsf{LF}})\delta$
SLINK^i	t_{SA}	$t_{\mathsf{SA}} f(\log^2 f + \log\log n)$	$t_{\mathsf{SA}} + (t_\psi + t_{\mathsf{LF}})\delta$
SDEP	t_{SA}	$t_{\mathsf{SA}} f \log^2 f$	$t_\psi \delta$
SLAQ	$t_{\mathsf{SA}} \log n$	$t_{\mathsf{SA}} f \log n \log^2 f$	$(t_\psi + t_{\mathsf{LF}})\delta$

CSA. For a node v represented as $[v_l, v_r]$ and PARENT(v) represented as $[v'_l, v'_r]$, if $v_l \neq v'_l$ then v_l is the index that represents v, otherwise we use v_r. The ROOT is represented by 0. So this identifier is computed in $O((t_\psi + t_{\mathsf{LF}})\delta)$ time, and guarantees that no index represents more than one node (as only the highest node of a leftmost/rightmost path can use the shared v_l/v_r value), but some indexes may represent no node at all. More precisely, this scheme yields identifiers in the range $[0, n-1]$ for the internal nodes, whereas there are only $t - n < n$ of them.

6. UPDATING THE SUFFIX TREE AND ITS SAMPLING

The static FCST requires that we first build the classical suffix tree and then sample it. Thus the machine used for construction must have a very large main memory, or we must resort to secondary memory suffix tree construction. Dynamic FCSTs permit handling a text collection where queries are interleaved with insertions and deletions of texts along time, and their space is asymptotically the same as their static variant. In particular, they solve the problem of construction of the static FCST within asymptotically the same space of the final static FCST: Start with an empty text collection, insert T, and then turn all the data structures into their static equivalents.

Along the paper we have given static and dynamic variants of all the data structures we have introduced. What remains is to explain how to modify our suffix tree representation to reflect the changes caused by inserting and removing texts T, and how to maintain our sampling conditions upon updates.

The CSA of Mäkinen and Navarro [2008], on which we build, inserts T in right-to-left order. It first determines the position of the new terminator and then uses LF to find the consecutive positions of longer and longer suffixes, until the whole T is inserted.[11] This right-to-left method perfectly matches with the algorithm by Weiner [1973] to build the suffix tree of T: It first inserts suffix $T[i+1..]$ and then suffix

[11]This insertion point is arbitrary in that CSA, thus there is no order among the texts. Moreover, all the terminators are the same in the CSA, yet this can be easily modified to handle different terminators as required in some bioinformatic applications [Gusfield 1997].

$T[i..]$, finding the points in the tree where the node associated to the new suffix is to be created if it does not already exist. The node is found by using PARENT until the WEINERLINK operation returns a non-empty interval. This requires one PARENT and one WEINERLINK amortized operation per symbol of T. This algorithm has the important invariant that the intermediate data structure is a suffix tree. Hence, by carrying it out in synchronization with the CSA insertion algorithm and with the insertion of the new leaves in bitvector B, we can use the current CSA and FCST to implement PARENT and WEINERLINK.

To maintain the property that the intermediate structure is a suffix tree, deletion of a text T must proceed by first locating the node of $\mathcal{T}$ that corresponds to T,[12] and then using SLINKs to remove all the nodes corresponding to its suffixes in $\mathcal{T}$. We must simultaneously remove the leaves in the CSA and in bitmap B (Mäkinen et al.'s CSA deletes a text right-to-left, but it is easy to adapt to use ψ instead of LF to do it left-to-right).

6.1. Maintaining the Sampling

We now explain how to update the sampled tree S whenever nodes are inserted into or deleted from the (virtual) suffix tree $\mathcal{T}$. The sampled tree must maintain, at all times, the property that for any node v there is an $i < \delta$ such that $\text{SLINK}^i(v)$ is sampled. The following concept from Russo and Oliveira [2008] is useful to explain how to obtain this result.

Definition 6.1. The *reverse tree* $\mathcal{T}^R$ of a suffix tree $\mathcal{T}$ is the minimal labeled tree that, for every node v of $\mathcal{T}$, contains a node v^R denoting the reverse string of the path label of v.

We note we are *not* maintaining nor sampling $\mathcal{T}^R$, we just use it as a conceptual device. Figure 2 shows a reverse tree. Observe that, since there is a node with path label ab in $\mathcal{T}$, there is a node with path label ba in $\mathcal{T}^R$. We can therefore define a mapping R that maps every node v to v^R. Observe that for any node v of $\mathcal{T}$, except for the ROOT, we have that $\text{SLINK}(v) = R^{-1}(\text{PARENT}(R(v)))$. This mapping is partially shown in Figures 1 and 2 by the numbers. Hence the reverse tree stores the information of the suffix links. For a regular sampling we choose the nodes for which $\text{TDEP}(v^R) \equiv_{\delta/2} 0$ and $\text{HEIGHT}(v^R) \geq \delta/2$. This is equivalent to our sampling rules on $\mathcal{T}$ (Theorem 4.2): Since the reverse suffixes form a prefix-closed set, $\mathcal{T}^R$ is a non-compact trie, *i.e.*, each edge is labeled by a single letter. Thus, $\text{SDEP}(v) = \text{TDEP}(v^R)$. The rule for $\text{HEIGHT}(v^R)$ is obviously related to that on $\text{SLINK}(v)$ by R. See Figure 2 for an example of this sampling.

Notice that, whenever a node is inserted or removed from a suffix tree, it never changes the SDEP of the other nodes in the tree, hence it does not change any TDEP in $\mathcal{T}^R$. This means that whenever the suffix tree is modified the only nodes that can be inserted or deleted from the reverse tree are the leaves. In $\mathcal{T}$ this means that when a node is inserted it does not break a chain of suffix links; it is always added at the beginning of such a chain. Weiner's algorithm works precisely by appending a new leaf to a node of $\mathcal{T}^R$.

Assume that we are using Weiner's algorithm and decide that the node $X.v$ should be added and we know the representation of node v. All we need to do to update the structure of the sampled tree is to verify whether by adding $(X.v)^R$ as a child of v^R in

[12]The dynamic CSA by Mäkinen and Navarro [2008] provides this functionality by returning a *handle* when inserting a text T, that can be used later to retrieve the CSA position of its first or last symbol. This requires $O(N \log n)$ extra bits of space when handling a collection of N texts and total length n, which is negligible unless one has to handle many short texts.

$\mathcal{T}^R$ we increase the $\textsc{Height}$ of some ancestor, in $\mathcal{T}^R$, that will now become sampled. Hence we must scan upwards in $\mathcal{T}^R$ to verify if this is the case. Also, we do not need to maintain $\textsc{Height}$ values. Instead, if the distance from $(X.v)^R$ to the closest sampled node $(v')^R$ is exactly $\delta/2$ and $\textsc{TDep}((v')^R) \equiv_{\delta/2} 0$, then we know that v' meets the sampling condition and we sample it. Operationally, the procedure is as follows: Compute $v' = \textsc{SLink}^{\delta/2}(v.X)$; if v' is not in S ($v' \neq \textsc{LSA}(v')$) and $\textsc{SDep}(v') \equiv_{\delta/2} 0$, then add v' to S.

Deleting a node (*i.e.*, a leaf in $\mathcal{T}^R$) is slightly more complex and involves some reference counting. This time assume we are deleting node $X.v$, again we need to scan upwards, this time to decide whether to make a node non-sampled. However $\textsc{SDep}(v) - \textsc{SDep}(v') < \delta/2$ is not enough, as it may be that $\textsc{Height}(v'^R) \geq \delta/2$ because of some other descendant. Therefore every sampled node v' counts how many descendants it has at distance $\delta/2$. A node becomes non-sampled (*i.e.*, we remove it from S) only when this counter reaches zero. Insertions and deletions of nodes in $\mathcal{T}$ must update these counters, by increasing/decreasing them whenever inserting/deleting a leaf at distance exactly $\delta/2$ from sampled nodes.

As there are $O(n/\delta)$ sampled nodes, reference counters count also sampled nodes, and no sampled node can be counted in two counters, we have that the sum of all the counters is also $O(n/\delta)$. Hence we represent them all using a bitmap C of $O(n/\delta)$ bits. C stores a 1 associated to each $\textsc{Preorder}_S(v)$, and that 1 is followed by as many 0s as the value of the counter for v. Hence the value of the counter for v is retrieved as $\textsc{Select}_1(C, \textsc{Preorder}_S(v) + 1) - \textsc{Select}_1(C, \textsc{Preorder}_S(v)) - 1$; increasing the counter for v translates into $\textsc{Insert}(C, \textsc{Select}_1(C, \textsc{Preorder}_S(v)) + 1, 0)$; and decreasing the counter into $\textsc{Delete}(C, \textsc{Select}_1(C, \textsc{Preorder}_S(v)) + 1)$. Similarly, insertion of a new node v into S must be followed by operation $\textsc{Insert}(C, \textsc{Select}_1(C, \textsc{Preorder}_S(v)), 1)$, and its deletion must be preceded by $\textsc{Delete}(C, \textsc{Select}_1(C, \textsc{Preorder}_S(v)))$. Using Theorem 2.4, structure C takes $O(n/\delta)$ bits and carries out all these operations in $O(\log n/\log\log n)$ time.

Hence, to $\textsc{Insert}$ or $\textsc{Delete}$ a node requires $O((t_\psi + t_{\mathsf{LF}})\delta)$ time to find out whether to modify the sampling, plus $O(\log n/\log\log n)$ time to update S and associated structures when necessary (S itself, B, C, etc.), plus $O(\log n)$ time to modify the sampled A and A^{-1} arrays. Added to the constant amortized number of calls to $\textsc{Parent}$ and $\textsc{WeinerLink}$ per text symbol, we have an overall time of $O(|T|(\log n + (t_\psi + t_{\mathsf{LF}})\delta))$ for the insertion or deletion of the whole text T.

The following theorem summarizes our result.

THEOREM 6.2. *It is possible to represent the suffix tree of a dynamic text collection within the space and time bounds given for DFCST in Table VI, by using any dynamic compressed suffix array offering the operations and times given in Table IV and inserting (deleting) texts in right-to-left (left-to-right) order.*

Table VI also compares our DFCST with the DCST of Chan et al. [2007]. For the latter we have used the faster dynamic trees of Theorem 2.6. There exists no dynamic variant of the EBST.

6.2. Changing $\log n$

We note that Theorem 6.2 assumes that $\lceil \log n \rceil$ is fixed, and so is δ. This assumption is not uncommon in dynamic data structures, even if it affects assertions like that of pointers taking $O(\log n)$ bits. The CSAs used in Table IV can handle varying $\lceil \log n \rceil$ within the same worst-case space and complexities, and the same happens with Theorem 2.4, which is used for bitmaps B and C, and with other data structures described by Navarro and Sadakane [2010] that we use for storing $\textsc{SDep}$ and by Mäkinen and

Table VI. Comparison between dynamic compressed suffix tree representations. The performance refers to dynamic CSA times and assumes $t_\psi, t_{\mathsf{LF}}, t_{\mathsf{SA}} = \Omega(t_{\mathsf{tree}})$; likewise we assume $\delta = \omega(\log n)$ as before. We omit the operations that depend solely on the CSA, see Table IV.

	Chan et al. [2007] (DCST)	Ours (DFCST)
Space in bits	$\lvert CSA\rvert + \mathbf{\Theta(n)}$	$\lvert CSA\rvert + O((n/\delta)\log n)$
ROOT	$O(1)$	$O(1)$
COUNT	t_{tree}	$O(1)$
ANCESTOR	t_{tree}	$O(1)$
PARENT	t_{tree}	$(t_\psi + t_{\mathsf{LF}})\delta$
FCHILD	t_{tree}	$t_{\mathsf{SA}} \log\delta + (t_\psi + t_{\mathsf{LF}})\delta + O(\log^2 n)$
NSIB	t_{tree}	$t_{\mathsf{SA}} \log\delta + (t_\psi + t_{\mathsf{LF}})\delta + O(\log^2 n)$
LCA	t_{tree}	$(t_\psi + t_{\mathsf{LF}})\delta$
CHILD	$t_{\mathsf{SA}} \log\sigma$	$t_{\mathsf{SA}} \log\delta + (t_\psi + t_{\mathsf{LF}})\delta + O(\log^2 n)$
SLINK	t_ψ	$(t_\psi + t_{\mathsf{LF}})\delta$
SLINK^i	t_{SA}	$t_{\mathsf{SA}} + (t_\psi + t_{\mathsf{LF}})\delta$
SDEP	t_{SA}	$t_\psi\delta$
INSERT(T) / DELETE(T)	$\lvert T\rvert(t_{\mathsf{SA}} + \log n)$	$\lvert T\rvert(t_\psi + t_{\mathsf{LF}})\delta$

Navarro [2008] that we use for handling the sampling of $A[j\theta]$ and $A^{-1}[j\theta]$ needed for CHILD. The dynamic parentheses data structure of Theorem 2.6 we use to represent S also allows for changes in $\lceil\log n\rceil$, but our mechanism to adapt to changes in δ will subsume it. We discuss now how to cope with this while retaining the same space and worst-case time complexities.

We use $\delta = \lceil\log n\rceil \cdot \lceil\log\lceil\log n\rceil\rceil$, which will change whenever $\lceil\log n\rceil$ changes (sometimes it will change by more than 1). Let us write $\delta = \Delta(\ell) = \ell\lceil\log\ell\rceil$. We maintain $\ell = \lceil\log n\rceil$. As S is small enough, we can afford to maintain three copies of it: S sampled with δ, S^- with $\delta^- = \Delta(\ell-1)$, and S^+ sampled with $\delta^+ = \Delta(\ell+1)$. When $\lceil\log n\rceil$ increases (*i.e.*, n doubles), S^- is discarded, the current S becomes S^-, the current S^+ becomes S, we build a new S^+ sampled with $\Delta(\ell+2)$, and ℓ is increased. A symmetric operation is done when $\lceil\log n\rceil$ decreases (*i.e.*, n halves due to deletions), so let us focus on increases from now on. Note this can occur in the middle of the insertion of a text, which must be suspended, and then resumed over the new set of sampled trees.

The construction of the new S^+ can be done by retraversing all the suffix tree $\mathcal{T}$ and deciding which nodes to sample according to the new δ^+. An initially empty parentheses sequence and a bitmap B^+ initialized with t zeros would give the correct insertion points from the chosen intervals as both structures are populated. To ensure that we consider each node of $\mathcal{T}$ once, we process the leaves in order (*i.e.*, $v = [0,0]$ to $v = [n-1, n-1]$), and for each leaf v we also consider all of its ancestors $[v_l, v_r]$ (using PARENT) as long as $v_r = v$. For each node $[v_l, v_r]$ we consider, we apply SLINK at most δ^+ times until we find the first node $v' = \text{SLINK}^i([v_l, v_r])$ which either is sampled in S^+, or $\text{SDEP}(v') \equiv_{\delta^+/2} 0$ and $i = \delta^+/2$. If v' was not sampled we insert it into S^+, and in both cases we increase its reference count (recall Section 6). All the δ^+ suffix links $\text{SLINK}^i([v_l, v_r])$ are computed in $O((t_\psi + t_{\mathsf{LF}})\delta^+) = O((t_\psi + t_{\mathsf{LF}})\delta)$ time, as they form a single chain.

Deamortization can be achieved by the classical method of interleaving the normal operations of the data structure with the construction of the new S^+. By performing a constant number of operations on the new S^+ for each insertion/deletion operation over the text collection, we can ensure that the new S^+ will be ready in time. We start by the creation (split into several operations) of B^+ formed by t 0s, and then proceed to traverse $\mathcal{T}$ to determine which nodes to insert into S. The challenge is to maintain the consistency of the traversal of $\mathcal{T}$ while texts are inserted/deleted.

As we insert a text, the operations that update $\mathcal{T}$ consist of insertion of leaves, and possibly creation of a new parent for them. Assume we are currently at node $[v_l, v_r]$

in our traversal of $\mathcal{T}$ to update S^+. If a new node $[v'_l, v'_r]$ we are inserting is behind the current node in our traversal order (that is, $v'_r < v_r$, or $v'_r = v_r$ and $v'_l > v_l$), then we consider $[v'_l, v'_r]$ immediately; otherwise we leave it for the moment when we will reach $[v'_l, v'_r]$ in our traversal (note that we will reach it in our traversal because the CSA has already been updated). Recall from Section 6 that those new insertions do not affect the existing SDEPs nor suffix link paths, and hence cannot affect the decisions to sample nodes already made in the current traversal. Similarly, deleted nodes that fall behind the current node are processed immediately, and the others are left for the traversal to handle them later.

If ℓ decreases again while we are still building S^+, we simply discard it even before having completed its construction. This involves freeing the whole B^+ and S^+ data structures, which is also necessary when we abandon the former B^- and S^- structures. This deallocation can be done in constant time in this particular case: The maximum size n the collection can have as long as we keep using the current S^+ and B^+ structures is $n_{max} = 2^{2+\lceil \log n \rceil}$, thus the maximum value for t is $t_{max} = 2n_{max}$ and for s is $s_{max} = t_{max}/(\delta^+/2)$. Hence, we allocate a single chunk of memory of the maximum possible size, which is still $O((n/\delta)\log \delta)$. A similar preallocation can be done for S^+, which needs $O(n/\delta)$ bits.

6.3. Construction from a Static CSA

An important side-effect of Theorem 6.2 is that operation INSERT(T) can be used to construct an FCST representation from scratch. We transform the DFCST into the FCST by changing the dynamic bitmaps (Theorem 2.4) into static ones (Theorem 2.3). This construction procedure works within $|CSA| + O((n/\delta)\log n)$ space, but it is time consuming. Using the dynamic CST in Table VI, and assuming $\sigma = O(\text{polylog}(n))$, the construction procedure takes $O(n \log^2 n)$ time.

It is possible to build the FCST faster from a static CSA. We simulate on the CSA a depth-first traversal on the reverse tree $\mathcal{T}^R$: Start at the ROOT and at each node v compute LF(X, v), for every $X \in \Sigma$. This process requires $O(n\sigma)$ time to traverse the reverse tree (we return to this soon). When LF(X, v) produces an interval with only one suffix, the traversal finishes because it corresponds to a leaf in the suffix tree. During this search we compute SDEP incrementally, as SDEP(LF$(X, v)) = 1 +$ SDEP(v), and we can easily compute the HEIGHT of each node v in $\mathcal{T}^R$ when we return to it during the recursive traversal. Thus the conditions that define which nodes to sample (Theorem 4.2) are easy to verify on the fly. The nodes to sample are stored in a list as pairs of integers $[v_l^{(i)}, v_r^{(i)}]$, therefore they require $O((n/\delta)\log n) = o(n)$ bits.

From this list we obtain the PREORDER representation of the sampled tree and the bitmap B of Section 4.1. We duplicate every pair $[v_l^{(i)}, v_r^{(i)}]$ in the list into a left pair $[v_l^{(i)}, v_r^{(i)}]_L$ and a right pair $[v_l^{(i)}, v_r^{(i)}]_R$. The list is then sorted. The sorting key for left pairs is $v_l^{(i)}$; in case of a tie it is $v_r^{(i)}$. The sorting key for right pairs is $v_r^{(i)}$; in case of a tie it is $v_l^{(i)}$. This means that if i corresponds to a left pair and i' to a right pair we will compare $v_l^{(i)}$ with $v_r^{(i')}$; note that for the crossed comparisons there are no ties. This procedure requires $O((n/\delta)\log(n/\delta)) = o(n)$ time. To obtain PREORDER$_S$ we traverse the list and write '(' for every left pair and ')' for every right pair. The bitmap B is obtained sequentially in a similar way: Consider the sequence of numbers o_i obtained by storing $v_l^{(i)}$ for the left nodes and $v_r^{(i)}$ for the right nodes. For every o_i, if $o_i = o_{i+1}$ we skip from o_i to o_{i+1}, otherwise we write a 1 in B followed by $o_{i+1} - o_i$ zeros. Recall that B finishes with 1 because the ROOT is always sampled.

Overall this construction procedure requires $O(n\sigma)$ time to obtain a static FCST representation from a static CSA. If the CSA is an FM-index implemented with a wavelet tree [Mäkinen and Navarro 2008], the time can be reduced to $O(n \log \sigma)$, as we can determine each existing interval $\text{LF}(X, v) = v'$ in time $O(\log \sigma)$. The reason is that the X symbols that produce valid intervals correspond to the distinct symbols in $T^{bwt}[v_l, v_r]$, where T^{bwt} is the Burrows-Wheeler transform of T [Burrows and Wheeler 1994], and if T^{bwt} is represented as a wavelet tree then we can use a so-called range quantile algorithm to list the distinct values in any substring within $O(\log \sigma)$ time per delivered value [Gagie et al. 2009].

Indeed, CSAs can be built from scratch faster than CSTs. For example, the FM-index can be built within $nH_k + o(n \log \sigma)$ bits and time $O(n \frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$ [Navarro and Sadakane 2010]. After making this CSA static we can build our CST, within negligible $o(n)$ extra space and negligible $O(n \log \sigma)$ extra time. There are algorithms to build the FM-index that take $o(n \log \sigma)$ time [Okanohara and Sadakane 2009; Hon et al. 2009], yet using $\Omega(n \log \sigma)$ construction space. Another intermediate solution is by Hon et al. [2003], using $O(n \log n)$ time and $O(nH_0)$ bits of space.

7. EXPERIMENTAL RESULTS

We implemented a basic version of the static FCST representation. We compare our prototype with an implementation of Sadakane's CST [Välimäki et al. 2007]. Our prototype uses a binary wavelet tree where each bitmap is encoded as in Theorem 2.3 (using an easier-to-implement proposal [Raman et al. 2002]); this CSA requires $nH_k + o(n \log \sigma)$ bits [Mäkinen and Navarro 2008]. The sampling factor δ was chosen as $\lceil \log n \rceil \cdot \lceil \log \lceil \log n \rceil \rceil$. We made a simple implementation that uses pointers in the sampled tree S, since the extra space requirement is still sub-linear. To minimize the amount of information stored in the sampled tree we chose not to support the TDEP, SLAQ and TLAQ operations, moreover for the same reason we chose to support only the basic CHILD operation and not the elaborated scheme presented in Section 5.2. One important difference between our implementation and the theory we presented is that the leaves of $\mathcal{T}$ are never part of the sampled tree S. This simplification is possible because $\text{LCA}(v, v')$ is a leaf only if $v = v'$, in which case $\text{LCA}(v, v') = v$ and $\text{SDEP}(\text{LCA}(v, v'))$ can be obtained from $A[v]$. Hence, the sampled tree becomes much smaller than in theory, as the sampled values of A are already considered as part of the CSA.

We used the texts from the Pizza&Chili corpus[13] trimmed to at most 100 megabytes (MB).

— Sources (program source code). This file is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions.
— Pitches (MIDI pitch values). This file is a sequence of pitch values (bytes in 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on Internet. The MIDI files were processed using semex 1.29 tool by Kjell Lemström, so as to convert them to IRP format. This is a human-readable tuple format, where the 5th column is the pitch value. Then the pitch values were coded in one byte each and concatenated.
— Proteins (protein sequences). This file is a sequence of newline-separated protein sequences (without descriptions, just the bare proteins) obtained from the Swissprot database. Each of the 20 amino acids is coded as one uppercase letter.

[13]http://pizzachili.dcc.uchile.cl

Table VII. Space requirements, in MB, of FCSTs and CSTs. The space is obtained by reporting peak of main memory usage when operating the data structures. Other related information such as the number of nodes in the sampled tree S and in $\mathcal{T}$ is also presented. Finally, we give data on relative space usage of the different components.

	Sources	Pitches	Proteins	DNA	English	XML
σ	229	134	26	18	217	98
$n/2^{20}$	100.0	53.2	63.7	100.0	100.0	100.0
$\|\mathcal{T}\|/2^{20}$	162.7	87.5	98.7	167.4	161.7	147.4
δ	135	130	130	135	135	135
$\|\mathcal{T}\|/\|S\|$	1,368	551	1,304	12,657	541	2,008
FCST (MB)	66.3	45.2	49.8	56.7	73.2	54.9
CST (MB)	407.4	214.3	204.2	287.2	353.7	316.3
$CSA_F/FCST$	0.90	0.88	0.92	0.92	0.87	0.91
CSA_C/CST	0.29	0.30	0.30	0.21	0.29	0.36
CSA_F/CSA_C	0.50	0.62	0.75	0.86	0.62	0.44

— DNA (DNA sequences). This file is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project. Each of the 4 bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special characters.
— English (English texts). This file is the concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. We deleted the headers related to the project so as to leave just the real text.
— XML (structured text). This file is an XML that provides bibliographic information on major computer science journals and proceedings and it is obtained from dblp.uni-trier.de.

We built FCSTs and CSTs for each of the previous files. The resulting space usage and related information is given in Table VII. The line "$n/2^{20}$" gives the file size in MB. We also count the number of nodes in each suffix tree in line "$|\mathcal{T}|/2^{20}$". It is interesting to observe that in practice the sampling rate of the internal nodes is much higher than δ (as several suffix link paths share the same sample). This can be observed by looking at line "$|\mathcal{T}|/|S|$": The ratio is usually 5 to 10 times larger than δ, but it reaches 93 times for DNA. The consequence of such a small sampling is that the percentage of our CSA size (CSA_F) in the overall structure is around 90%, see line "$CSA_F/FCST$".

Lines FCST and CST show that our FCST is 4 to 6 times smaller than the CST. This is a consequence not only of the fact that the size (CSA_F) of our CSA is only 44% to 86% the size (CSA_C) of the CSA used by the CST implementation (see line CSA_F/CSA_C), but more importantly, that our tree structure occupies a much smaller portion of the overall space than in the CST, see lines CSA_C/CST and $CSA_F/FCST$. Hence, in terms of space we managed to obtain an extremely compact representation of FCSTs. Moreover the fact that our implementation uses pointers increases the overall space by only a negligible amount of space.

Overall, our structure takes 55% to 85% of the original text size and moreover *replaces it*, as the CSA itself can reproduce any substring of the sequence. Thus, our representation can be regarded as a *compressed representation* of the sequence which, in addition, provides a suffix tree functionality on it. We now consider how time-efficient is this functionality.

We tested the time it takes to compute the operations in Theorem 6.2 by choosing internal nodes, computing the operations during 60 seconds, and obtaining averages per operation. We used three ways of choosing the nodes to test the operations. To select a node we chose a random leaf v and computed LCA$(v, v+1)$. We used three sequences of random nodes. In the first case we chose only one random node as described (u). In the second case we chose a random node and iterated SLINK (su) until reaching the root, collecting all the traversed nodes. In the last case we chose a random node and iterated PARENT (pu) until reaching the root. This simulates various types of suffix tree traversals. The results are shown in table VIII. Our machine had a Quad-Core Intel Xeon CPU at 3.20GHz with a 2 MB cache, 3 GB of RAM, and was running Slackware 12.0.0 with Linux kernel 2.6.21.5. The FCSTs were implemented in C and compiled with `gcc 3.4.6 -O9`. The CSTs were implemented in C++ and compiled with `g++ 4.1.2 -O3`.

The results show that the price for our FCST's small space requirements is that they are much slower than CSTs, yet practical in absolute terms for many applications (i.e., few milliseconds per operation). For some operations such as LCA the diference can reach 3 orders of magnitude. Still for the CHILD operation, which is the slowest, the diference is usually 1 order of magnitude. Hence, in any algorithm that uses CHILD, this operation should dominate the overall time, moreover it depends essentially on the underlying CSA. We expect this to be the case in general. Therefore it is possible to obtain diferent space time trade-offs by using other CSAs.

Our implementation aimed at obtaining the smallest possible FCSTs. The resulting space/time trade-off is interesting because we obtained very small FCSTs that support the usual operations within a reasonable time per operation. Recently published experiments [Cánovas and Navarro 2010] that compare the performance of a practical implementation of the EBST with the CST and FCST reinforce the conclusion that our FCST, albeit being the slowest of the three, is unparalleled in space requirements, which makes it able to fit in main memory suffix trees that no other representation can handle.

8. LARGER AND FASTER COMPRESSED SUFFIX TREES

The previous discussion raises the question of whether it is possible to obtain better times from our technique, perhaps using more space. In particular, we note that using a smaller δ value in our FCST would yield better times. What prevents us from using values smaller than $\delta = \log n \log\log n$ is that we have to spend $O((n/\delta)\log n) = O(\frac{n}{\log\log n})$ extra bits. However, this space comes only from the storage of SDEP and TDEP arrays[14].

Imagine we use *both* the FM-index and the sublinear-space CSA by Grossi et al. [2003], for a total space of $(2+\frac{1}{\epsilon})nH_k + o(n\log\sigma)$ bits, so that we have $t_\psi = t_{\mathsf{LF}} = O(1+\frac{\log\sigma}{\log\log n})$, and $t_{\mathsf{SA}} = O(t_\psi \log_\sigma^\epsilon n)$. Now we could store only the SDEP values at nodes whose SDEP is a multiple of κ, and at the other sampled nodes v we only store $\text{SDEP}(v) \bmod \kappa$ using $\log\kappa$ bits. The total space for SDEP becomes $O((n/\kappa)\log n + (n/\delta)\log\kappa)$. To retrieve a $\text{SDEP}(v)$ value, we read $d = \text{SDEP}(v) \bmod \kappa$, and then read the full $c = \text{SDEP}(v')$, where $v' = \text{SLINK}^d(v)$ has its full SDEP value stored. The answer is $c+d$ and can be obtained in $O(t_{\mathsf{SA}})$ time[15]. The same idea can be used for TDEP, which is stored for tree depths multiple of κ and retrieved using $v' = \text{PARENT}^i(v)$ in time $O(\log n)$.[16] Now, we can use $\kappa = \log n \log\log n$ and $\delta = (\log\log n)^2$ while maintaining the extra space $O(\frac{n}{\log\log n})$. Although we use a much smaller δ now, each step requires computing a SDEP value in $O(t_{\mathsf{SA}})$ time, and thus our usual $(t_\psi + t_{\mathsf{LF}})\delta$ cost becomes $t_{\mathsf{SA}}\delta = O(\log_\sigma^\epsilon n(\log\sigma + \log\log n)\log\log n)$. If $\sigma = \text{polylog}(n)$, this can be written

[14]We have been careful along the paper to avoid this type of space for the other data structures, which could otherwise have been handled with classical solutions.
[15]Since we know v' is a sampled node, we do $v' = \text{LCSA}(\psi^i(v_l), \psi^i(v_r))$ without resorting to LCA, which would have implied a circular dependence.
[16]Again, because v' is sampled it can be binary searched for in $\text{PARENT}_S^j(v)$.

Table VIII. Time to compute operations over the FCST (F) and CST (C), in seconds.

Operation			Sources	Pitches	Proteins	DNA	English	XML
LCA	F	u	6.9e-3	8.2e-3	2.9e-3	2.0e-3	6.3e-3	9.7e-3
	F	su	1.3e-2	2.0e-2	7.7e-3	2.2e-3	1.9e-2	7.6e-3
	F	pu	4.3e-3	9.0e-3	1.0e-3	1.1e-3	1.6e-3	4.0e-3
	C	u	2.5e-6	2.5e-6	2.3e-6	2.2e-6	2.4e-6	2.4e-6
	C	su	2.5e-6	2.1e-6	2.7e-6	4.2e-6	1.9e-6	3.5e-6
	C	pu	5.4e-6	5.1e-6	5.0e-6	5.7e-6	5.6e-6	5.7e-6
Letter	F	u	1.6e-5	1.6e-5	1.1e-5	8.9e-6	1.3e-5	1.5e-5
	F	su	1.8e-5	1.6e-5	1.1e-5	8.4e-6	1.4e-5	1.5e-5
	F	pu	1.5e-5	1.4e-5	9.9e-6	8.4e-6	1.3e-5	1.4e-5
	C	u	1.6e-4	1.3e-4	1.2e-4	7.1e-5	1.4e-4	1.6e-4
	C	su	7.4e-5	7.1e-5	7.4e-5	5.0e-5	8.2e-5	9.8e-5
	C	pu	1.1e-4	6.8e-5	8.8e-5	5.8e-5	1.3e-4	1.4e-4
SLink	F	u	6.8e-3	8.2e-3	2.9e-3	2.0e-3	6.2e-3	9.6e-3
	F	su	1.3e-2	2.0e-2	7.7e-3	2.1e-3	1.9e-2	7.6e-3
	F	pu	4.3e-3	9.0e-3	9.6e-4	1.0e-3	1.6e-3	3.9e-3
	C	u	2.2e-4	1.7e-4	1.7e-4	9.6e-5	2.0e-4	2.2e-4
	C	su	1.1e-4	9.6e-5	1.0e-4	6.7e-5	1.1e-4	1.4e-4
	C	pu	1.7e-4	1.0e-4	1.5e-4	8.4e-5	1.9e-4	2.0e-4
Locate	F	u	3.2e-3	3.0e-3	1.8e-3	1.6e-3	2.7e-3	3.1e-3
	F	su	3.0e-3	2.7e-3	1.7e-3	1.3e-3	2.6e-3	2.9e-3
	F	pu	2.7e-3	2.3e-3	1.6e-3	1.3e-3	2.4e-3	2.6e-3
	C	u	5.0e-5	3.9e-5	3.8e-5	2.2e-5	4.4e-5	5.0e-5
	C	su	2.2e-5	1.8e-5	2.0e-5	1.5e-5	2.0e-5	2.7e-5
	C	pu	3.7e-5	2.1e-5	3.0e-5	1.9e-5	4.3e-5	4.3e-5
Child	F	u	8.2e-3	1.3e-2	3.4e-3	1.6e-3	1.1e-2	8.7e-3
	F	su	1.9e-2	3.2e-2	1.0e-2	3.7e-3	3.8e-2	6.5e-3
	F	pu	6.5e-3	1.6e-2	9.9e-4	7.7e-4	1.9e-3	3.3e-3
	C	u	5.8e-4	5.4e-4	4.2e-4	1.2e-4	5.2e-4	6.4e-4
	C	su	2.0e-4	2.6e-4	2.8e-4	1.2e-4	1.3e-4	1.2e-3
	C	pu	2.5e-3	1.6e-3	9.0e-4	1.9e-4	2.9e-3	2.2e-3
SDep	F	u	5.4e-3	6.6e-3	2.3e-3	1.6e-3	5.2e-3	7.3e-3
	F	su	1.1e-2	1.8e-2	5.9e-3	1.7e-3	1.7e-2	5.7e-3
	F	pu	4.0e-3	7.8e-3	8.0e-4	8.4e-4	1.3e-3	3.0e-3
	C	u	5.1e-5	4.0e-5	3.8e-5	2.3e-5	4.5e-5	5.0e-5
	C	su	2.1e-5	1.8e-5	2.0e-5	1.5e-5	2.0e-5	2.6e-5
	C	pu	3.6e-5	2.1e-5	3.3e-5	2.2e-5	4.1e-5	4.4e-5
Parent	F	u	8.3e-3	8.7e-3	2.6e-3	3.4e-3	4.3e-3	1.5e-2
	F	su	1.3e-2	7.6e-3	5.0e-3	2.5e-3	4.5e-3	1.3e-2
	F	pu	6.3e-3	1.3e-2	1.1e-3	1.8e-3	2.1e-3	6.1e-3
	C	u	1.6e-6	1.7e-6	1.7e-6	1.6e-6	1.6e-6	1.7e-6
	C	su	1.5e-6	1.5e-6	1.6e-6	1.6e-6	1.6e-6	1.7e-6
	C	pu	1.5e-6	1.5e-6	1.6e-6	1.6e-6	1.6e-6	1.7e-6

as $O(\log^{\epsilon} n)$. Thus we achieve sublogarithmic times for most operations. Indeed the times are similar to those of the EBST and our space is better than their original version [Fischer et al. 2009], yet their most recent result [Fischer 2010] achieves better space.

We can go further and achieve poly-loglog times for the most common operations, at the expense of higher space. We use their representation for LCP information that gives constant-time access and $2nH_k(\log \frac{1}{H_k} + O(1)) + o(n)$ bits of space [Fischer et al. 2009]. Recall that $\text{LCP}(v) = \text{SDep}([v-1, v])$ is the longest common prefix between leaves v and $v-1$. In addition they show how to compute range minimum queries $\text{RMQ}(v, v')$ (which gives the minimum value in the range $\text{LCP}(v) \ldots \text{LCP}(v')$) using, for example, $O(\frac{n}{\log \log n})$ bits of space and $O(\log \log n (\log \log \log n)^2)$ time. Using this we can obtain directly $\text{SDep}([v_l, v_r]) = \text{RMQ}(v_l + 1, v_r)$. The same method can be applied for TDep. Now the only limit to decrease δ is array B, which uses $O((n/\delta) \log \delta)$ bits, and this is $o(n)$ for any $\delta = \omega(1)$. Yet, let us restrict to $O(\frac{n}{\log \log n})$ extra space, so we use $\delta = \log\log n \log\log\log n$. If we use an FM-index as our CSA, our final CST size is $2nH_k(\log \frac{1}{H_k} + O(1)) + o(n)$ bits, and our usual $(t_\psi + t_{\mathsf{LF}})\delta$ time for most operations becomes $O(\log\log n(\log\sigma + \log\log n)(\log\log\log n)^3)$. This is $o((\log\log n)^3)$ for $\sigma = O(\mathrm{polylog}(n))$.

9. CONCLUSIONS

We presented a fully-compressed representation of suffix trees (FCSTs), which breaks the linear-bit space barrier of previous representations at a reasonable time complexity penalty. Our structure efficiently supports common and not-so-common operations, including very powerful ones such as lowest common ancestor (LCA) and level ancestor (LAQ) queries. Indeed, by building over an FM-index, our FCSTs achieve optimal asymptotic space under the k-th order entropy model, and support all the navigation operations in polylogarithmic time. Our representation is largely based on the LCA operation. Suffix trees have been used in combination with LCAs for a long time, but our results show new ways to exploit this partnership. We also presented a dynamic fully-compressed representation of suffix trees. Dynamic FCSTs permit not only managing dynamic collections, but also building static FCSTs within optimal space, at a logarithmic time penalty factor.

We implemented a static version of the FCSTs and showed that its surprisingly small space requirements can be obtained in practice and it can still support the usual operations efficiently. A recent experimental comparison [Cánovas and Navarro 2010] of compressed suffix trees confirms that the FCST is the smallest representation, albeit it is also the slowest. Using a denser sampling on our current implementation does not give interesting space/time tradeoffs, but we are pursuing a new one where such a denser sampling makes a better impact on response times.

The research on this topic advances at a very rapid pace. In the last two years, after the conference publication of our results [Russo et al. 2008b; 2008a], several new achievements have been presented. The progress was mainly focused on obtaining smaller representations of the data structures that support Range Minimum Queries (RMQs), and the so-called Previous Smaller Value (PSV) and Next Smaller Value (NSV) queries. The results by Ohlebusch et al. [2009; 2010] reduced the constants associated with the $O(n)$-bit space term. Although the resulting space is still $\Theta(n)$, they achieve relevant improvements. An implementation of the EBST [Fischer et al. 2009] also provided new practical techniques to implement RMQ/PSV/NSV operations [Cánovas and Navarro 2010], as well as the mentioned experimental comparison among different prototypes. Fischer [2010] improved the original EBST [Fischer et al. 2009] by removing the "ugly" space factor associated to the entropy, that is, the new EBST now requires $(1 + \frac{1}{\epsilon})nH_k + o(n)$ bits and retains the same sublogarithmic time performance (we used this improved complexity in our Table I).

The techniques we introduce in this paper also have demonstrated to have independent interest. Recently, Hon et al. [2009] improved the secondary memory index proposed by Chien et al. [2008] using, among other techniques, a structure similar to the bitmap B we presented in Section 4.1.

We believe this fascinating topic is far from closed. In particular, we have exposed limitations for some operations on FCSTs, which might or might not be fundamental. For example we give only a partial answer to the problem of computing the preorder number of a suffix tree node, which is relevant to associate satellite information to internal nodes. Another important example is the lack of support for the TDEP, TLAQ, and SLAQ operations on dynamic FCSTs. This has its roots in our inability to maintain a properly spaced sampling of the suffix tree, and maintain TDEP values up to date. Yet a third example are the limitations on the alphabet size σ in order to have

$o(n)$ extra space. Our prototype is also being extended to support the dynamic case and, as mentioned, denser samplings.

More generally, and especially under the light of the combinations of ideas explored in the previous section, it is not clear how fast can we navigate suffix trees using how much space, and in general which is the space/time lower bound for compressed suffix trees.

A APPENDIX

In this appendix we explore some fundamental properties of suffix trees that show how to use a δ-sampled suffix tree. Section 5 makes use of these properties to provide the different navigation operations, albeit it can be read without resorting to this deeper discussion.

More specifically, we reveal some *self-similarity* properties of suffix trees. Such properties have already been studied, but as far as we know, the ones we study here are novel.

1.1. Locally Isomorphic Subtrees

Gusfield [1997, Section 7.7] showed that suffix trees contain isomorphic subtrees. This type of information is useful because storing a self-similar structure naively contains redundant information. Therefore self-similarities can be exploited to remove the redundant information in the representation of suffix trees. Gusfield used this property to define compact DAGs that are similar, but distinct, to the earlier concept of DAWGs [Blumer et al. 1985] and CDAWGs [Crochemore 1986].

An isomorphism is a bijective homomorphism. An homomorphism from a tree to another tree is a function that preserves, in some way, the structure of the source tree in the target tree. The type of homomorphism we use depends on the algebraic structure that we consider. For example, we can consider that an homomorphism between trees $\mathcal{T}_1$ and $\mathcal{T}_2$ is a function f, of the nodes, for which $f(\text{PARENT}(v_1)) = \text{PARENT}(f(v_1))$ for any node v_1 of $\mathcal{T}_1$. For this notion, graph-homomorphism, Gusfield presented the following lemma:

LEMMA 1.1 (1.1 [GUSFIELD 1997]). *If the number of leaves of the subtree below v is equal to the number of leaves below $\text{SLINK}(v)$, then the two subtrees are graph-isomorphic.*

The isomorphism is given by the SLINK function. Reproving this lemma will be useful to introduce the new concepts. We only need to show that SLINK is injective, surjective and structure-preserving. It is interesting to notice that the SLINK function is not globally injective, for example $\text{SLINK}(ab) = b = \text{SLINK}(bb)$. However by restricting its domain it becomes injective.

LEMMA 1.2. *The SLINK function from the subtree below any node $v \neq \text{ROOT}$ is injective.*

PROOF. Since $v \neq \text{ROOT}$ we have that the path label of v is $X.\alpha$ where $X \in \Sigma$ and $\alpha \in \Sigma^*$. Let v' and v'' be descendants of v, *i.e.*, with path labels $X.\alpha.\beta'$ and $X.\alpha.\beta''$ respectively, such that $\text{SLINK}(v') = \text{SLINK}(v'')$. This means that $\alpha.\beta = \alpha.\beta'$ and therefore $X.\alpha.\beta = X.\alpha.\beta'$ which means that $v' = v''$. □

With an extra condition SLINK becomes surjective.

LEMMA 1.3. *If the number of leaves of the subtree below node $v \neq \text{ROOT}$ is equal to the number of leaves below $\text{SLINK}(v)$, then SLINK is surjective.*

PROOF. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ denote the subtrees below v and SLINK(v) respectively. The proof consists in showing that $|\mathcal{T}_1| \geq |\mathcal{T}_2|$, *i.e.*, $\mathcal{T}_2$ has no more nodes than $\mathcal{T}_1$. This implies that SLINK is surjective, since, by Lemma 1.2, SLINK is injective.

We denote the number leaves of a tree $\mathcal{T}'$ by $\lambda(\mathcal{T}')$. It is easy to prove, by induction, that for any tree $\mathcal{T}'$ the following property holds:

$$|\mathcal{T}'| = 2 * \lambda(\mathcal{T}') - 1 - \sum_{\text{internal } v' \in \mathcal{T}'} (-2 + \text{number of children of } v)$$

Hence, since $\lambda(\mathcal{T}_1) = \lambda(\mathcal{T}_2)$, all we need to show is that $\sum_{v_1 \in \mathcal{T}_1}(\ldots) \leq \sum_{v_2 \in \mathcal{T}_2}(\ldots)$. Note that the terms of the sum are always non-negative because $\mathcal{T}_1$ and $\mathcal{T}_2$ are compact, *i.e.*, have at least two children. Since SLINK is injective this result can be shown directly by arguing that the number of children of the internal node v_1 in $\mathcal{T}_1$ is not larger than the number of children of SLINK(v_1) in $\mathcal{T}_2$. This is a known property of suffix trees: If node v_1 contains a child node that branches by letter $X \in \Sigma$, then SLINK(v_1) must also contain a child branching by X. SLINK does not remove these letters from the path label because v_1 descends from $v \neq$ ROOT. □

To complete the proof of Lemma 1.1 we still need the following property, whose proof we postpone to the next subsection, where we will have more algebraic tools.

LEMMA 1.4. *If the number of leaves of the subtree below node $v \neq$ ROOT is equal to the number of leaves below* SLINK(v) *then it holds* SLINK(PARENT(v')) = PARENT(SLINK(v')) *for any node v' descendant of v.*

The compact DAG data structure [Gusfield 1997] removes the regularity arising from Lemma 1.1 by storing pointers from the node v to the node SLINK(v), whenever v satisfies the conditions of the lemma.

1.2. Locally Monomorphic Subtrees

The problem with the regularities exploited by the DAG approach is that the associated similarity concept is too strict. In this section we consider the relation between the subtrees below v and SLINK(v) under less regular conditions, *i.e.*, when the number leaves below v and SLINK(v) is different. Obviously SLINK will not be surjective, but the proof of Lemma 1.2 is still valid and therefore SLINK is still locally injective. Moreover, SLINK will no longer be a graph homomorphism, although it is still preserving the tree structure in some way. This means we need a new notion of homomorphism that is less strict than the one related to PARENT. Hence we will now consider trees as being partially ordered sets (posets), ordered by the ANCESTOR relation. A poset homomorphism has the following definition:

Definition 1.5. A tree poset-homomorphism, between trees $\mathcal{T}_1$ and $\mathcal{T}_2$, is a mapping f from the nodes of $\mathcal{T}_1$ to the nodes of $\mathcal{T}_2$, such that if ANCESTOR(v', v) holds in $\mathcal{T}_1$, then ANCESTOR$(f(v'), f(v))$ holds in $\mathcal{T}_2$.

By reasoning with the path labels it should be obvious that SLINK is always a homomorphism in this sense. In fact SLINK is also a homomorphism in a slightly more regular way. We presented the poset-homomorphism because it gives a more intuitive description of the structure being preserved. With the LCA operation trees can also be considered as semilattices, *i.e.*, for any nodes v, v' and v'' we have LCA$(v, v) = v$; LCA(v, v') = LCA(v', v); LCA$(v,$ LCA$(v', v''))$ = LCA(LCA$(v, v'), v'')$. The poset-homomorphisms are not necessarily semilattice-homomorphisms, although they have the following property:

LEMMA 1.6. *If there is a tree poset-homomorphism f between trees $\mathcal{T}_1$ and $\mathcal{T}_2$, then for any nodes v and v' of $\mathcal{T}_1$ we have that $f(\mathrm{LCA}(v, v'))$ is an ancestor of* $\mathrm{LCA}(f(v), f(v'))$.

PROOF. The proof consists in showing that $f(\mathrm{LCA}(v, v'))$ is an ancestor of both $f(v)$ and $f(v')$, since by definition of LCA this guarantees that it must be an ancestor of $\mathrm{LCA}(f(v), f(v'))$. Let us consider $f(v)$, without loss of generality. Since $\mathrm{LCA}(v, v')$ is an ancestor of v we conclude from Definition 1.5 that $f(\mathrm{LCA}(v, v'))$ is an ancestor of $f(v)$. □

Lemma 5.1 points out that in fact SLINK is a much more special kind of homomorphism: It is a semilattice-homomorphism, and that is essentially the underlying insight of our suffix tree representation technique. In light of this property it is now very easy to prove Lemma 1.4.

PROOF OF LEMMA 1.4. The proof follows from computing the expression:

$$\mathrm{SLINK}(\mathrm{LCA}(\mathrm{PARENT}(v), \mathrm{SLINK}^{-1}(\mathrm{PARENT}(\mathrm{SLINK}(v))))).$$

Note that the expression is well defined because, under the conditions of the lemma, SLINK is bijective. First note that since $\mathrm{PARENT}(\mathrm{SLINK}(v))$ is an ancestor of $\mathrm{SLINK}(v)$, $\mathrm{SLINK}^{-1}(\mathrm{PARENT}(\mathrm{SLINK}(v)))$ is an ancestor of $\mathrm{SLINK}^{-1}(\mathrm{SLINK}(v)) = v$. Moreover, since SLINK is bijective and $\mathrm{PARENT}(\mathrm{SLINK}(v)) \neq \mathrm{SLINK}(v)$, we have that $\mathrm{SLINK}^{-1}(\mathrm{PARENT}(\mathrm{SLINK}(v))) \neq v$. This means that $\mathrm{SLINK}^{-1}(\mathrm{PARENT}(\mathrm{SLINK}(v)))$ is an ancestor of $\mathrm{PARENT}(v)$ and therefore it is the LCA we are trying to compute. Therefore the expression yields $\mathrm{SLINK}(\mathrm{SLINK}^{-1}(\mathrm{PARENT}(\mathrm{SLINK}(v)))) = \mathrm{PARENT}(\mathrm{SLINK}(v))$.

On the other hand we can try to compute our expression by using Lemma 5.1 and therefore obtain:

$$\mathrm{LCA}(\mathrm{SLINK}(\mathrm{PARENT}(v)), \mathrm{PARENT}(\mathrm{SLINK}(v)))$$

Now we use essentially the same reasoning, in a slightly simpler way. As $\mathrm{PARENT}(v)$ is an ancestor of v we have that $\mathrm{SLINK}(\mathrm{PARENT}(v))$ is an ancestor of $\mathrm{SLINK}(v)$. Moreover, since SLINK is injective and $\mathrm{PARENT}(v) \neq v$, it holds $\mathrm{SLINK}(\mathrm{PARENT}(v)) \neq \mathrm{SLINK}(v)$. Thus $\mathrm{SLINK}(\mathrm{PARENT}(v))$ is an ancestor of $\mathrm{PARENT}(\mathrm{SLINK}(v))$ and therefore it is the LCA we are trying to compute.

Hence we obtained that $\mathrm{PARENT}(\mathrm{SLINK}(v)) = \mathrm{SLINK}(\mathrm{PARENT}(v))$. □

Note that, although the first relation in the proof is only true under the conditions of Lemma 1.4, the fact that $\mathrm{SLINK}(\mathrm{PARENT}(v))$ is an ancestor of $\mathrm{PARENT}(\mathrm{SLINK}(v))$ is always true.

Hence we have shown that, when $v \neq \mathrm{ROOT}$, the SLINK operation is a monomorphism, *i.e.*, injective homomorphism, that preserves LCA. *This means that suffix trees are locally self-similar as semilattices.* This regularity, as far-fetched as it may be, is very important because any representation of suffix trees that ignores it will contain redundant information. The regularity implies, roughly, that the subtree below $\mathrm{SLINK}(v)$ contains the subtree below v, and that we can use LCA to recover it. This is the conceptual basis of Lemma 5.2.

REFERENCES

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. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

ARROYUELO, D. 2008. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th International Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 5029. 277–289.

BLUMER, A., BLUMER, J., HAUSSLER, D., EHRENFEUCHT, A., CHEN, M., AND SEIFERAS, J. 1985. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science 40,* 1, 31–55.

BURROWS, M. AND WHEELER, D. 1994. A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation.

CÁNOVAS, R. AND NAVARRO, G. 2010. Practical Compressed Suffix Trees. In *Proc. 9th International Symposium on Experimental Algorithms (SEA)*. LNCS 6049. 94–105.

CHAN, H.-L., HON, W.-K., AND LAM, T.-W. 2004. Compressed index for a dynamic collection of texts. In *Proc. 15th International Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 3109. 445–456.

CHAN, H.-L., HON, W.-K., LAM, T.-W., AND SADAKANE, K. 2007. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms 3,* 2, article 21.

CHIEN, Y.-F., HON, W.-K., SHAH, R., AND VITTER, J. S. 2008. Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In *Proc. Data Compression Conference (DCC)*. 252–261.

CROCHEMORE, M. 1986. Transducers and repetitions. *Theoretical Computer Science 45,* 1, 63–86.

FERRAGINA, P. AND MANZINI, G. 2000. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*. 390–398.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed text. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed Representations of Sequences and Full-Text indexes. *ACM Transactions on Algorithms 3,* 2, article 20.

FERRAGINA, P. AND VENTURINI, R. 2007. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science 372,* 1, 115–121.

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.

FOSCHINI, L., GROSSI, R., GUPTA, A., AND VITTER, J. 2006. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms 2,* 4, 611–639.

GAGIE, T., PUGLISI, S. J., AND TURPIN, A. 2009. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th Symposium on String Processing and Information Retrieval (SPIRE)*. 1–6.

GIEGERICH, R., KURTZ, S., AND STOYE, J. 2003. Efficient implementation of lazy suffix trees. *Software Practice and Experience 33,* 11, 1035–1049.

GONZÁLEZ, R. AND NAVARRO, G. 2008. Rank/Select on Dynamic Compressed Sequences and Applications. *Theoretical Computer Science 410*, 4414–4422.

GROSSI, R., GUPTA, A., AND VITTER, J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

GUPTA, A., HON, W.-K., SHAH, R., AND VITTER, J. 2007. A framework for dynamizing succinct data structures. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*. LNCS 4596. 521–532.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press.

HE, M. AND MUNRO, I. 2010. Succinct representations of dynamic strings. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 6393. 334–346.

HON, W.-K., LAM, T.-W., SADAKANE, K., AND SUNG, W.-K. 2003. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*. 240–249.

HON, W. K., SADAKANE, K., AND SUNG, W. K. 2009. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM Journal on Computing 38,* 6, 2162–2178.

HON, W.-K., SHAH, R., THANKACHAN, S., AND VITTER, J. 2009. On Entropy-Compressed Text Indexing in External Memory. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 75–89.

HUYNH, T. N. D., HON, W.-K., LAM, T. W., AND SUNG, W.-K. 2006. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science 352,* 1-3, 240–249.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996a. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing*. 141–155.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996b. Sparse suffix trees. In *Computing and Combinatorics*. LNCS 1090. 219–230.

LEE, S. AND PARK, K. 2007. Dynamic Rank-Select Structures with Applications to Run-Length Encoded Texts. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4580. 95–106.

MÄKINEN, V. AND NAVARRO, G. 2008. Dynamic Entropy-Compressed Sequences and Full-Text Indexes. *ACM Transactions on Algorithms 4,* 3, 1–38.

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

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, I., RAMAN, V., AND RAO, S. S. 2001. Space efficient suffix trees. *Journal of Algorithms 39*, 205–222.

NAVARRO, G. 2004. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms 2,* 1, 87–114.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed Full-Text Indexes. *ACM Computing Surveys 39,* 1, article 2.

NAVARRO, G. AND SADAKANE, K. 2010. Fully-functional static and dynamic succinct trees. *CoRR* abs/0905.0768. `http://arxiv.org/abs/0905.0768`. Version 4.

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

OHLEBUSCH, E. AND GOG, S. 2009. A compressed enhanced suffix array supporting fast string matching. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 51–62.

OKANOHARA, D. AND SADAKANE, K. 2009. A linear-time burrows-wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 90–101.

PĂTRAŞCU, M. AND VIOLA, E. 2010. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 117–122.

PĂTRAŞCU, M. 2008. Succincter. In *Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 305–313.

RAMAN, R., RAMAN, V., AND RAO, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 233–242.

RAMAN, R. AND RAO, S. S. 2003. Succinct Dynamic Dictionaries and Trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*. LNCS 2719. 357–368.

RUSSO, L., NAVARRO, G., AND OLIVEIRA, A. 2008a. Dynamic Fully-Compressed Suffix Trees. In *Proc. 19th International Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 5029. 191–203.

RUSSO, L., NAVARRO, G., AND OLIVEIRA, A. 2008b. Fully-Compressed Suffix Trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*. LNCS 4957. 362–373.

RUSSO, L. M. S. AND OLIVEIRA, A. L. 2008. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval 11,* 4, 359–388.

SADAKANE, K. 2003. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms 48,* 2, 294–313.

SADAKANE, K. 2007. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems 41,* 4, 589–607.

SADAKANE, K. AND NAVARRO, G. 2010. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 134–149.

VÄLIMÄKI, N., GERLACH, W., DIXIT, K., AND MÄKINEN, V. 2007. Engineering a compressed suffix tree implementation. In *Proc. 6th International Workshop on Efficient and Experimental Algorithms (WEA)*. LNCS 4525. 217–228.

WEINER, P. 1973. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory (SWAT)*. 1–11.

Received Month Year; revised Month Year; accepted Month Year