# Dynamic Fully-Compressed Suffix Trees

Luís M. S. Russo[⋆⋆⋆1,3], Gonzalo Navarro[†2], and Arlindo L. Oliveira[1]

[1] INESC-ID, R. Alves Redol 9, 1000 Lisboa, Portugal. aml@algos.inesc-id.pt
[2] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
[3] Dept. of Computer Science, University of Lisbon, Portugal. lsr@di.fc.ul.pt

**Abstract.** Suffix trees are by far the most important data structure in stringology, with myriads of applications in fields like bioinformatics, data compression and information retrieval. Classical representations of suffix trees require $O(n \log n)$ 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 $\sigma$ is the alphabet size. The size of suffix trees has been a barrier to their wider adoption in practice. A recent so-called fully-compressed suffix tree (FCST) requires asymptotically only the space of the text entropy. FCSTs, however, have the disadvantage of being static, not supporting updates to the text. In this paper we show how to support dynamic FCSTs within the same optimal space of the static version and executing all the operations in polylogarithmic time. In particular, we are able to build the suffix tree within optimal space.

## 1 Introduction and Related Work

Suffix trees are extremely important for a large number of string processing problems. Their many virtues have been described by Apostolico [1] and Gusfield [2]. The combinatorial properties of suffix trees have a profound impact in the *bioinformatics* field, which needs to analyze large 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 to be useful in practice, renders them inapplicable in the cases where they would be most useful, that is, on large texts.

The space problem is so important that it originated a plethora of research results, ranging from space-engineered implementations [3] to novel data structures that simulate suffix trees, most notably suffix arrays [4]. Some of those space-reduced variants give away some functionality in exchange. For example suffix arrays miss the important suffix link navigational operation. Yet, all these classical approaches require $O(n \log n)$ bits, while the indexed string requires only $n \log \sigma$ bits (we write log for $\log_2$), $n$ being the size of the string and $\sigma$ the

⋆⋆⋆ Supported by the Portuguese Science and Technology Foundation by grant SFRH/BPD/34373/2006 and project ARN, PTDC/EIA/67722/2006.

† Partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

size of the alphabet. For example the human genome requires 700 Megabytes, while even a space-efficient suffix tree on it requires at least 40 Gigabytes [5], and the reduced-functionality suffix array requires more than 10 Gigabytes. This is particularly evident in DNA because $\log \sigma = 2$ is much smaller than $\log n$.

These representations are also much larger than the size of the *compressed* string. Recent approaches [6] combining data compression and succinct data structures have achieved spectacular results for the pattern search problem. For example Ferragina *et al.* [7] presented an index that requires $nH_k + o(n \log \sigma)$ bits and counts the occurrences of a pattern of length $m$ in time $O(m(1 + (\log_\sigma \log n)^{-1}))$. Here $nH_k$ denotes the $k$-th order empirical entropy of the string [8], 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)[4], and, by adding a few extra structures, support all the operations provided by suffix trees. Sadakane presented the first *compressed suffix tree* (CST) [5], adding $6n$ bits on top of the CSA. Recently Russo *et al.* [9] achieved a *fully-compressed suffix tree* (FCST), which works over the smallest existing CSA [7], adding only $o(n \log \sigma)$ bits to it. Hence the FCST breaks the $\Theta(n)$ extra-bits space barrier and retains asymptotic space optimality.

Albeit very interesting as a first step, the FCST has the limitation of being static, and moreover of being built from the uncompressed suffix tree. CSAs have recently overcome this limitation, starting with the structure by Chan *et al.* [10]. In its journal version this work included the first dynamic CST, which builds on Sadakane's (static) CST [5] and retains its $\Theta(n)$ extra space penalty. On the other hand, the smallest existing CSA [7] was made dynamic within the same space by Mäkinen *et al.* [11], which was recently improved by González *et al.* [12] so as to achieve logarithmic time slowdown. In this paper we make the FCST dynamic by building on this latter dynamic CSA. We retain the optimal space complexity and polylogarithmic time for all the operations.

A comparison between Chan *et al.*'s CST and our FCST is shown in Table 1. Our FCST is not significantly slower, yet it requires much less space (e.g. one can realistically predict 25% of Chan *et al.*'s CST space on DNA). For the table we chose the smallest existing dynamic CSA, so that we show the time complexities that can be obtained within the smallest possible space for both CSTs.

All these dynamic structures, as well as ours, indeed 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.

## 2 Basic Concepts

Fig. 1 illustrates the concepts in this section. We denote by $T$ a **string**; by $\Sigma$ the **alphabet** of size $\sigma$; by $T[i]$ the symbol at position ($i \bmod n$) (so the first symbol

[4] These are also called compact suffix arrays, FM-indexes, etc., see [6].

**Table 1.** Comparing compressed suffix tree representations. The operations are defined along Section 2. Time complexities, but not space, are big-O expressions. We give the generalized performance (assuming $\Psi, t, \Phi \geq \log n$) and an instantiation using $\delta = (\log_\sigma \log n) \log n$. For the instantiation we also assume $\sigma = O(\text{polylog}(n))$, and use the dynamic FM-Index variant of González *et al.* [12] as the compressed suffix array ($CSA$), for which the space holds for any $k \leq \alpha \log_\sigma(n) - 1$ and any constant $0 < \alpha < 1$.

| | Chan *et al.* [10] | Ours |
|---|---|---|
| Space in bits | $\|CSA\| + \mathbf{O(n)} + o(n)$ $= nH_k + \mathbf{O(n)} + o(n \log \sigma)$ | $\|CSA\| + O((n/\delta) \log n)$ $= nH_k + o(n \log \sigma)$ |
| SDep | $\Phi = (\log_\sigma \log n) \log^2 n$ | $\Psi\delta = (\log_\sigma \log n) \log^2 n$ |
| Count/ Ancestor | $\log n = \log n$ | $1 = 1$ |
| Parent | $\log n = \log n$ | $(\Psi + t)\delta = (\log_\sigma \log n) \log^2 n$ |
| SLink | $\Psi = \log n$ | $(\Psi + t)\delta = (\log_\sigma \log n) \log^2 n$ |
| SLink$^i$ | $\Phi = (\log_\sigma \log n) \log^2 n$ | $\Phi + (\Psi + t)\delta = (\log_\sigma \log n) \log^2 n$ |
| Letter / Locate | $\Phi = (\log_\sigma \log n) \log^2 n$ | $\Phi = (\log_\sigma \log n) \log^2 n$ |
| LCA | $\log n = \log n$ | $(\Psi + t)\delta = (\log_\sigma \log n) \log^2 n$ |
| FChild/ NSib | $\log n = \log n$ | $(\Psi + t)\delta + \Phi \log \delta + (\log n) \log(n/\delta)$ $= ((\log_\sigma \log n) \log^2 n) \log \log n$ |
| Child | $\Phi \log \sigma = (\log \log n) \log^2 n$ | $(\Psi + t)\delta + \Phi \log \delta + (\log n) \log(n/\delta)$ $= ((\log_\sigma \log n) \log^2 n) \log \log n$ |
| WeinerLink | $t = \log n$ | $t = \log n$ |
| Insert($T$) / Delete($T$) | $\|T\|(\Psi + t)\delta$ $= \|T\|(\log_\sigma \log n) \log^2 n$ | $\|T\|(\Psi + t)\delta = \|T\|(\log_\sigma \log n) \log^2 n$ |

is $T[0]$); by $T.T'$ **concatenation**; by $T = T[..i-1].T[i..j].T[j+1..]$ respectively a **prefix**, a **susbtring** and a **suffix**; by Parent$(v)$ the parent node of node $v$; by TDep$(v)$ its tree-depth; by Ancestor$(v, v')$ whether $v$ is an ancestor of $v'$; by LCA$(v, v')$ the **lowest common ancestor**.

The **path-label** of a node $v$ in a labeled tree is the concatenation of the edge-labels from the root down to $v$. We refer indifferently to nodes and to their path-labels, also denoted by $v$. The $i$-th letter of the path-label is denoted as Letter$(v, i) = v[i]$. The **string-depth** of a node $v$, denoted by SDep$(v)$, is the length of its path-label. Child$(v, X)$ is the node that results of descending from $v$ by the edge whose label starts with symbol $X$, if it exists. The **suffix tree** of $T$ is the deterministic compact labeled tree for which the path-labels of the leaves are the suffixes of $T$. We assume that $T$ ends in a terminator symbol \$ that does not belong to $\Sigma$. The **generalized suffix tree** of a collection $\mathcal{C}$ of texts is the tree that results from merging the respective suffix trees. Moreover each text is assumed to have a distinct terminator. For a detailed explanation see Gusfield's book [2]. The **suffix-link** of a node $v \neq$ Root of a suffix tree, denoted SLink$(v)$, is a pointer to node $v[1..]$. Note that SDep$(v)$ of a leaf $v$ identifies the suffix of $T$ starting at position $n-$SDep$(v) =$ Locate$(v)$. For example $T[\text{Locate}(ab\$)..] = T[7-3..] = T[4..] = ab\$$. The **suffix array** $A[0, n-1]$ stores the Locate values of the leaves in lexicographical order. Note that in a generalized suffix

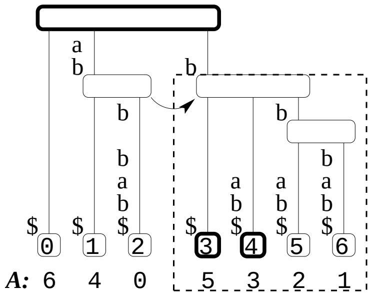


**Fig. 1.** Suffix tree $\mathcal{T}$ of string *abbbab*, with the leaves numbered. The arrow shows the SLINK between node *ab* and *b*. Below it we show the suffix array. The portion of the tree corresponding to node *b* and respective leaves interval is within a dashed box. The sampled nodes have bold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

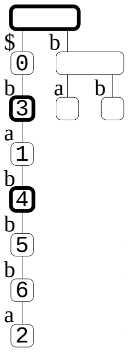

**Fig. 2.** Reverse tree $\mathcal{T}^R$.

```
                    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.

tree LOCATE must also identify the text to which the suffix corresponds. When we use arithmetic expressions involving $A$ and $A^{-1}$ they are computed within a given text, *i.e.* they do not jump to another text. Moreover for simplicity we use only one text in our example and hence omit the text identifier. The *suffix tree nodes can be identified with suffix array intervals*: each node corresponds to the *range* of leaves that descend from $v$. The node $b$ corresponds to the interval $[3, 6]$. Hence the node $v$ will be represented by the interval $[v_l, v_r]$. Leaves are also represented by their left-to-right index (starting at 0). 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 COUNT in constant time the number of leaves that descend from $v$. The number of leaves below $b$ is $4 = 6 - 3 + 1$. This is precisely the number of times that the string $b$ occurs in the indexed string $T$. We can also compute ANCESTOR in $O(1)$ time: $\text{ANCESTOR}(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$.

## 3 Static Fully-Compressed Suffix Trees and Our Plan

In this section we briefly explain the static FCST we build on [9]. The FCST consists of a compressed suffix array, a $\delta$-sampled tree $S$, and mappings between these structures. We also give the road map of our plan to dynamize the FCST.

**Compressed suffix arrays (CSAs)** are compact and functional representations of suffix arrays [6]. Apart from the basic functionality of retrieving $A[i] = \text{LOCATE}(i)$ (within a time complexity that we will call $\Phi = \Omega(\log n)$), state-of-the-art CSAs 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 $\Psi = \Omega(\log n)$. The iterated version of $\psi$, denoted $\psi^i$, can usually be computed faster than $O(i\Psi)$ with CSAs. This is achieved as $\psi^i(v) = A^{-1}[A[v]+i]$, let us assume that the CSA can also compute $A^{-1}$ within $O(\Phi)$ time. CSAs might also support the $\textsc{WeinerLink}(v, a)$ operation [13]: for a node $v$ the $\textsc{WeinerLink}(v, X)$ gives the suffix tree node with path-label $X.v[0..]$. This is called the LF mapping in CSAs, and is a kind of inverse of $\psi$, let its time complexity be $t = \Omega(\log n)$. Consider the interval $[3, 6]$ that represents the leaves whose path-labels start by $b$. In this case we have that $\mathrm{LF}(a, [3, 6]) = [1, 2]$, *i.e.* by using the LF mapping with $a$ we obtain the interval of leaves whose path-labels start by $ab$. We extend of LF to strings, $\mathrm{LF}(X.Y, v) = \mathrm{LF}(X, \mathrm{LF}(Y, v))$.

CSAs also implement $\textsc{Letter}(v, i)$ for leaves $v$. The easiest case is the first letter of a given suffix, $\textsc{Letter}(v, 0) = T[A[v]]$. This corresponds to $v[0]$, the first letter of the path-label of leaf $v$. Dynamic CSAs implement $v[0]$ in time $O(\log n)$. In general, $\textsc{Letter}(v, i) = \textsc{Letter}(\textsc{SLink}^i(v), 0)$ is implemented in $O(\Phi)$ time. CSAs are usually self-indexes, meaning that they replace the text: they can extract any substring, of size $\ell$, of the indexed text in $O(\Phi + \ell\Psi)$ time.

In this paper we will use a dynamic CSA for this part [12], which implements these operations with logarithmic slowdown to its static version [7]. The dynamic CSA actually handles a collection of texts, where insertions and deletions of whole texts $T$ are carried out in time $O(|T|(\Psi + t))$.

**The $\delta$-sampled tree** exploits the property that suffix trees are self-similar, $\textsc{SLink}(\mathrm{LCA}(v, v')) = \mathrm{LCA}(\textsc{SLink}(v), \textsc{SLink}(v'))$ whenever the expressions are well defined. This means, roughly, that the tree structure below $\textsc{SLink}(v)$ contains the tree structure below $v$. Because of this regularity it is possible to store only a few sampled nodes instead of the whole suffix tree. A *$\delta$-sampled tree $S$*, from a suffix tree $\mathcal{T}$ of $\Theta(n)$ nodes, chooses $O(n/\delta)$ nodes such that, for each node $v$, node $\textsc{SLink}^i(v)$ is sampled for some $i < \delta$. Such a sampling can be obtained by choosing nodes with $\textsc{SDep}(v) \equiv_{\delta/2} 0$ such that there is another node $v'$ for which $v = \textsc{SLink}^{\delta/2}(v')$. For such a sampling Lemma 1 holds, where $\mathrm{LCSA}(v, v')$ is the *lowest common sampled ancestor* of $v$ and $v'$:

**Lemma 1.** *Let $v, v'$ be nodes such that $\textsc{SLink}^r(\mathrm{LCA}(v, v')) = \textsc{Root}$, and let $d = \min(\delta, r + 1)$. Then*
$\textsc{SDep}(\mathrm{LCA}(v, v')) = \max_{0 \le i < d}\{i + \textsc{SDep}(\mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v')))\}$.

By itself however this property leads to an entangled loop of operations, because LCA depends on $\textsc{SLink}$ and $\textsc{SLink}(v) = \mathrm{LCA}(\psi(v_l), \psi(v_r))$ depends on LCA. Using CSAs and observing that $\mathrm{LCA}(v, v') = \mathrm{LCA}(\min\{v_l, v'_l\}, \max\{v_r, v'_r\})$ we can simplify this equation to $\textsc{SDep}(\mathrm{LCA}(v, v')) = \max_{0 \le i < d}\{i + \textsc{SDep}(\mathrm{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\})))\}$.

Therefore the kernel operations can be computed as:

$\textsc{SDep}(v) = \textsc{SDep}(\mathrm{LCA}(v, v)) = \max_{0 \le i < d}\{i + \textsc{SDep}(\mathrm{LCSA}(\psi^i(v_l), \psi^i(v_r)))\}$

$\mathrm{LCA}(v, v') = \mathrm{LF}(v[0..i-1], \mathrm{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\})))$

from which $\textsc{SLink}$ is obtained as well. The $i$ in the last equation is the one that maximizes $\textsc{SDep}(\mathrm{LCA}(v, v'))$. Operation $\textsc{Parent}(v)$ is easily computed on top of LCA. These operations take time $O((\Psi + t)\delta)$, except that $\textsc{SDep}$ takes $O(\Psi\delta)$.

Note that we have to solve LCSA. This requires to solve $\mathrm{LCA}_S$, that is, LCA queries on the sampled tree $S$, and also to map nodes to sampled nodes using operation LSA (see later). The sampled tree also needs to solve $\textsc{Parent}_S$ and store $\textsc{SDep}_\mathcal{T}$. The rest is handled by the CSA.

For the dynamic version, we first show how the suffix tree $\mathcal{T}$ changes upon insertion and deletion of texts $T$ to the collection. Then we show how to maintain the sampling properties of $S$ under those updates of the (virtual) $\mathcal{T}$. This will require some more data to be stored in the sampled nodes. Finally, we will make use of a dynamic parentheses representation for the sampled tree, which will already give us $\mathrm{LCA}_S$ and $\textsc{Parent}_S$, as well as a way to associate data to nodes and insert/delete nodes. Note that we just have to show how to provide this basic tree functionality, as the remaining operations are obtained as in the static version.

To support TDEP, however, they add other $O(n/\delta)$ nodes to the sampling, such that for any node $v$ the node $\textsc{Parent}^j(v)$ is sampled, for some $0 \leq j < \delta$. We have not found a way to efficiently maintain this second sampling in a dynamic scenario. As a consequence, our dynamic FCST does not support operation TDEP nor those that require it [9]: LAQT and LAQS. The basic navigation operations FCHILD and NSIB also require TDEP, but we will present a different idea that solves them together with CHILD and a generalization of it, using just the CSA.

**Mapping between the CSA and the sampled tree.** For every node $v$ of the sampled tree we need to obtain the corresponding interval $[v_l, v_r]$. On the other hand, given a CSA interval $[v_l, v_r]$ representing node $v$ of $\mathcal{T}$, the *lowest sampled ancestor* $\mathrm{LSA}(v)$ gives the lowest sampled tree node containing $v$. With LSA we can compute $\mathrm{LCSA}(v, v') = \mathrm{LCA}_S(\mathrm{LSA}(v), \mathrm{LSA}(v'))$.

In this paper we introduce a new method to implement these mappings that is efficient and simpler than the one presented in the static version [9].

## 4 Updating the Suffix Tree and Its Sampling

In this section we explain how to modify a suffix tree to reflect changes caused by inserting and removing a text $T$ to/from the suffix tree.

The CSA of Mäkinen *et al.* [11], on which we build, inserts $T$ in right-to-left order. It first determines the position of the new terminator[5] and then uses LF to find the consecutive positions of longer and longer suffixes, until the whole $T$ is inserted. This right-to-left method perfectly matches with Weiner's algorithm [13] to build the suffix tree of $T$: it first inserts suffix $T[i+1..]$ and then suffix $T[i..]$, 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

[5] 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 it can be easily modified to handle different terminators.

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, we can use the current CSA 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 $T$, 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 (Mäkinen *et al.*'s CSA deletes a text right-to-left as well, but it is easy to adapt to use $\Psi$ instead of $LF$ to do it left-to-right).

We now explain how to update the sampled tree $S$ whenever nodes are inserted 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 *et al.* [14] is fundamental to explain how to obtain this result.

**Definition 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. Fig. 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 Figs. 1 and 2 by the numbers. Hence the reverse tree stores the information of the suffix links. By $\text{Height}(v^R)$ we refer to the distance between $v$ and its farthest descendant leaf. 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 $\mathcal{T}$ (Section 3): 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 Fig. 2 for an example of this sampling.

Likewise, stating that there is an $i < \delta$ for which $\text{SLink}^i(v)$ is sampled is the same as stating that there is an $i < \delta$ for which $\text{TDep}(\text{Parent}^i(v^R)) \equiv_{\delta/2} 0$ and $\text{Height}(\text{Parent}^i(v^R)) \geq \delta/2$ . Since $\text{TDep}(\text{Parent}^i(v^R)) = \text{TDep}(v^R) - i$, the first condition holds for exactly two $i$'s in $[0, \delta[$. Since Height is strictly increasing the second condition holds for sure for the largest $i$. Notice that since every sampled node has at least $\delta/2$ descendants that are not sampled, this means that we sample at most $\lfloor 4n/\delta \rfloor$ nodes from a suffix tree with $\leq 2n$ nodes.

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 that if by adding $(X.v)^R$ as a child of $v^R$ in $\mathcal{T}^R$ we increase the HEIGHT of some of 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. Notice that we already carry out this scanning as a side effect of computing SLINK($v$), which also gives us the required SDEP information. Also, we do not need to maintain HEIGHT values. Instead, if the distance from $(X.v)^R$ to the closest sampled node $(v')^R$ is exactly $\delta/2$ and $\text{TDEP}((v')^R) \equiv_{\delta/2} 0$, then we know that $v'$ meets the sampling condition and we sample it.

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 $\text{SDEP}(v) - \text{SDEP}(v') < \delta/2$ is not enough, as it may be that $\text{HEIGHT}(v'^R) \geq \delta/2$ 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 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 nodes.

Hence to INSERT or DELETE a node requires $O((\Psi + t)\delta)$ time, plus the time to manipulate the structure that holds the topology of $S$: we need to carry out insertions/deletions of nodes, while maintaining information associated to them (SDEP, reference counts). Section 5.2 shows that those operations do not dominate the time $O((\Psi + t)\delta)$ needed to maintain the sampling conditions.

## 5 Dynamic Fully-Compressed Suffix Trees

In this section we present the compact data structures we use and create to handle our dynamic structures: the CSA, the sampled tree, and the mappings.

### 5.1 Dynamic Compressed Suffix Arrays

To maintain a dynamic CSA we use the following result by González *et al.* [12], which is an improvement upon those of Mäkinen *et al.* [11]:

**Theorem 1.** *A dynamic CSA over a collection $\mathcal{C}$ of texts can be stored within $nH_k(\mathcal{C}) + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma(n) - 1$ and any constant $0 < \alpha < 1$, supporting all the operations with times $t = \Psi = O(((\log_\sigma \log n)^{-1} + 1) \log n)$, $\Phi = O((\log_\sigma \log n) \log^2 n)$, and inserting/deleting texts $T$ in time $O(|T|(t + \Psi))$.*

Note that for a collection with $p$ texts it is necessary to store the positions of the texts in $A$. This requires $O(p \log n)$ bits but it is not an issue unless the texts are very short [11].

Therefore, the problem of maintaining a dynamic CSA is already solved, except that we promised to support operation $\text{CHILD}_{\mathcal{T}}$ (and some derivatives) directly on the CSA. Indeed, $\text{CHILD}_{\mathcal{T}}(v, X)$ can be easily computed in $O(\Phi \log n)$ time by binary searching for the interval of $v = [v_l, v_r]$ formed by those $v'$ where $\text{LETTER}(v', \text{SDEP}(v) + 1) = X$. Similarly, $\text{FCHILD}(v)$ can be determined by computing $X = \text{LETTER}(v_l, \text{SDEP}(v) + 1)$ and then $\text{CHILD}_{\mathcal{T}}(v, X)$. To compute $\text{NSIB}(v)$ the process is similar: 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 $\text{CHILD}_{\mathcal{T}}(v, X)$. All the time complexities are thus dominated by that of $\text{CHILD}_{\mathcal{T}}$.

Now we show how $\text{CHILD}_{\mathcal{T}}$ can be computed in a more general and efficient way. 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 sub-interval of the $v'$'s such that $\psi^m(v') \in v_2$, where $m = \text{SDEP}(v_1)$. This approach requires $O(\Phi \log n)$ time and it was first considered using CSA's by Huynh *et al.* [15]. Thus we are able to generalize $\text{CHILD}_{\mathcal{T}}(v, X)$, which uses $v_2$ as the sub-interval of $A$ of the suffixes starting with $X$.

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

The overall process requires time $O(\Phi + (\Psi + t)\delta)$ to access the last letters of $v_1$ and build $D$, plus $O((\log n) \log(n/\delta))$ for binary searching the samples; plus $O(\Phi \log \delta)$ for the final binary searches. We have assumed $O(\log n)$ time to access the sampled $A$ and $A^{-1}$ values in a dynamic scenario, whereas in a static scenario[6] it would be $O(1)$.

In fact in a dynamic scenario we do not store exactly the $A[j\delta]$ values; instead we guarantee that for any $k$ there is a $k'$ such that $k - \delta < k' \le k$ 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. For example, Mäkinen *et al.* [11, Sec. 7.1 of journal version] describes 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 to maintain approximately spaced samples in $A$, which can be done exactly as for $A^{-1}$).

[6] This speedup immediatly improves the results of Huynh *et al.* [15].

### 5.2 Dynamic Sampled Trees

The sampled tree contains only $O(n/\delta)$ nodes. As such it could be stored with pointers using only $O((n/\delta)\log n)$ bits. Instead we use a dynamic parentheses data structure given by Chan *et al.* [10], which already supports LCA.

**Theorem 2.** *A list of $O(n/\delta)$ balanced parentheses can be maintained in $O(n/\delta)$ bits supporting the following operations in $O(\log n)$ time:*

- FindMatch$(u)$*, finds the matching parenthesis of $u$;*
- Enclose$(u)$*, finds the nearest pair of matching parentheses that encloses $u$;*
- DoubleEnclose$(u, u')$*, finds the nearest pair of parentheses that encloses both $u$ and $u'$;*
- Insert$(u, u')$*,* Delete$(u, u')$*, inserts or deletes the matching parentheses located at $u, u'$.*

The Enclose primitive computes Parent$_S$ in the sampled tree. Likewise the DoubleEnclose primitive computes the LCA$_S$ operation. In Section 5.3 we explain how to update the parentheses sequence when a node becomes sampled or non-sampled (*i.e.* , how to maintain the mapping with the CSA). Operations Rank and Select on the sequence of parentheses $S$ can also be used to store information on the nodes, by mapping between the parentheses sequence and their preorder values and vice versa: Rank$_{'('}(S, i)$ gives the preorder number of the node identified by the opening parenthesis at $S[i]$, while Select$_{'('}(S, j)$ identifies the $j$-th node (in preorder) in $S$. Rank and Select over the parentheses bitmap can be handled using the following theorem.

**Theorem 3 ([11]).** *A bitmap of $n$ bits supporting* Rank*,* Select*,* Insert *and* Delete *in $O(\log n)$ time can be maintained in $nH_0 + O(n/\sqrt{\log n})$ bits.*

Each node of $S$ must also store its SDep. This is not complicated because the SDep of the nodes of $\mathcal{T}$ does not change, at least using Weiner's algorithm. Thus we maintain a balanced tree where the SDep values can be read, inserted, and deleted, at the positions given by Rank$_{'('}(S, i)$. When a node becomes sampled/non-sampled we insert/delete in this sequence. A similar mechanism is used to store the reference counts used for the sampling; in this case the stored values can be modified as well. Thus $O(\log n)$ time suffices for simulating the tree operations on $S$.

### 5.3 Mapping from CSA to the Sampled Tree and Back

The *lowest sampled ancestor* LSA is the way to map from the CSA to $S$. LSA is computed by using an operation Reduce$(v)$, that receives the numeric representation of leaf $v$ and returns the position, in the parentheses representation of the sampled tree, where that leaf should be. Consider for example the leaf numbered 5 in Fig. 3. This leaf is not sampled, but in the original tree it appears somewhere between leaf 4 and the end of the tree, more specifically between parenthesis ')' of 4 and parenthesis ')' of the Root. We assume Reduce returns the first parenthesis, *i.e.* $\textsc{Reduce}(5) = 4$. In this case since the parenthesis we obtain is a ')' we know that LSA should be the parent of that node. Hence we compute LSA as follows:

$$\mathrm{LSA}(v) = \begin{cases} \textsc{Reduce}(v) & \text{, if } S[\textsc{Reduce}(v)] \;=\; '(' \\ \textsc{Parent}(\textsc{Reduce}(v)) & \text{, otherwise} \end{cases}$$

We present a new way to compute $\textsc{Reduce}$ in $O(\log n)$ time and $o(n)$ bits (cf. [9]). We use a bitmap $B$ initiated with $n$ bits all equal to 0. Now for every node $v = [v_l, v_r]$ we insert a 1 at $\textsc{Select}_0(B, v_l)$ and after $\textsc{Select}_0(B, v_r)$, which yields a bitmap with $n + O(n/\delta)$ bits. In our example it is 1000101101001, see Fig. 3. Hence we have the following relation $\textsc{Reduce}(v) = \textsc{Rank}_1(B, \textsc{Select}_0(B, v + 1)) - 1$. We do not store $B$ uncompressed, but rather using Theorem 3, which requires only $O((n/\delta) \log n)$ bits as there are few 1's in $B$. When a node $[v_l, v_r]$ becomes sampled we insert matching parentheses at $S[\textsc{Reduce}(v_l)]$ and after $S[\textsc{Reduce}(v_r)]$. Also, it is necessary to insert the new 1's in $B$ as before. Fig. 3 illustrates the effect of sampling $b = [3, 6]$.

Updating $S$ when a sampled node $v$ becomes non-sampled is easy, as we can obtain the parentheses $u$, $u'$ to delete. We must also delete the corresponding 1's in $B$; note that the relative position of a 1 in a run of 1's is irrelevant. Therefore $\textsc{Reduce}$ can be computed in $O(\log n)$ time. According to our previous explanation, so can LSA and LCSA, for leaves.

To map in the other direction, each node in the sampled tree must know its corresponding interval $[v_l, v_r]$. This is also easy to obtain from $B$. Let $u$ be the position in $S$ of the opening parenthesis that identifies sampled node $v$. The corresponding closing parenthesis is $u' = \textsc{FindMatch}(u)$. Now $v_l = \textsc{Rank}_0(B, \textsc{Select}_1(B, u + 1))$ and $v_r = \textsc{Rank}_0(B, \textsc{Select}_1(B, u' + 1)) - 1$.

## 6 Putting All Together

The following theorem summarizes our result.

**Theorem 4.** *It is possible to represent the suffix tree of a dynamic text collection within the space and time bounds given in Table 1. The space and the variables $\Psi$, $\Phi$, $t$, can be instantiated to the values of Theorem 1 for $\delta = \omega(\log_\sigma n)$, or to another dynamic CSA supporting $\psi$, $A$, $A^{-1}$, LF, and $T[A[v]]$, in times $O(\Psi)$, $O(\Phi)$, $O(\Phi)$, $O(t)$, and $O(\log n)$, respectively, provided texts are inserted in right-to-left order and deleted in left-to-right order within the given time bounds.*

We note that Theorem 4 assumes that $\lceil \log n \rceil$ is fixed, and so is $\delta$. This assumption is not uncommon in dynamic data structures, even if it affects assertions like that of pointers taking $O(\log n)$ bits. The CSA used in Theorem 1 can handle varying $\lceil \log n \rceil$ within the same worst-case space and complexities, and the same happens with Theorem 3, which is used for the mapping bitmap $B$. The only remaining part is the sampled tree. We discuss now how to cope with it while retaining the same space and worst-case time complexities.

We use $\delta = \lceil \log n \rceil \cdot \lceil \log_\sigma \lceil \log n \rceil \rceil$, which will change whenever $\lceil \log n \rceil$ changes (sometimes will change by more than 1). Let us write $\delta = \Delta(\ell) = \ell \lceil \log_\sigma \ell \rceil$. We maintain $\ell = \lceil \log n \rceil$. As $S$ is small enough, we can afford to maintain three copies of it: $S$ sampled with $\delta$, $S^-$ 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 $\ell$ 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 $\mathcal{T}$ deciding which nodes to sample according to the new $\delta^+$. An initially empty parentheses sequence and a bitmap $B$ initialized with 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 its ancestors $[v_l, v_r]$ (using $\text{Parent}_{\mathcal{T}}$) as long as $v_r = v$. For each node $[v_l, v_r]$ we consider, we apply SLink at most $\delta^+$ times until either 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 $i \geq \delta^+/2$. If $v'$ was not sampled we insert it into $S^+$, and in both cases we increase its reference count if $i = \delta^+/2$ (recall Section 4).

All the $\delta^+$ suffix links in $\mathcal{T}$ are computed in $O(\delta^+(\Psi + t))$ time, as they form a single chain. Therefore the solution maintains the current complexities, yet only in an amortized sense.

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 $\mathcal{C}$, we can ensure that the new $S^+$ will be ready in time. 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 inserted 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 this for the moment when we will reach $[v'_l, v'_r]$ in our traversal. Recall from Section 4 that those new insertions do not affect the existing SDeps nor suffix link paths, and hence can be considered independently of the current traversal process. Similarly, deleted nodes that fall behind the current node are processed immediately, and the others left for the traversal to handle it.

If $\ell$ decreases while we are still building $S^+$, we can discard it even before having completed its construction. Note that in general discarding a tree when $\ell$ changes involves freeing several data structures. This can also be done progressively, interleaved with the other operations.

## 7 Conclusions

We presented the first dynamic fully-compressed representation of suffix trees (FCSTs). Static FCSTs broke the $\Theta(n)$ bits barrier of previous representations at a reasonable (and in some cases no) time complexity penalty, while retaining a surprisingly powerful set of operations. Dynamic FCSTs permit not only managing dynamic collections, but also building static FCSTs within optimal space. Hence the way is open to practical implementations of this structure, which can run in main memory for very large texts.

We also gave some relevant results for the static case, as we improved or simplified the operations Reduce and Child. A challenge for future work is to obtain operations TDep, LAQt, and LAQs, which we were not able to maintain in a dynamic scenario.

*Acknowledgments.* We are grateful to Veli Mäkinen and Johannes Fisher for pointing out the generalized branching problem to us.

## References

1. Apostolico, A. Combinatorial Algorithms on Words. NATO ISI Series. In: The myriad virtues of subword trees. Springer-Verlag (1985) 85–96
2. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
3. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Softw., Pract. Exper. **33**(11) (2003) 1035–1049
4. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5) (1993) 935–948
5. Sadakane, K.: Compressed Suffix Trees with Full Functionality. Theo. Comp. Sys. (2007)
6. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. **39**(1) (2007) article 2
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algor. **3**(2) (2007) article 20
8. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3) (2001) 407–430
9. Russo, L., Navarro, G., Oliveira, A.: Fully-compressed suffix trees. In: LATIN. LNCS (2008)
10. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Trans. Algorithms **3**(2) (2007)
11. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. In: CPM. LNCS 4009 (2006) 307–318 To appear in *ACM TALG*.
12. González, R., Navarro, G.: Improved dynamic rank-select entropy-bound structures. In: Proc. LATIN. LNCS (2008)
13. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. on Switching and Automata Theory. (1973) 1–11
14. Russo, L., Oliveira, A.: A compressed self-index using a Ziv-Lempel dictionary. In: Proc. SPIRE. LNCS 4209 (2006) 163–180
15. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. Theor. Comput. Sci. **352**(1-3) (2006) 240–249