Fully-Compressed Suffix Trees

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

[1] INESC-ID, R. Alves Redol 9, 1000 LISBOA, PORTUGAL
{lsr,aml}@algos.inesc-id.pt
[2] Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. Suffix trees are by far the most important data structure in stringology, with myriads of applications in fields like bioinformatics 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 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 still unsatisfactory when σ is small as in DNA sequences.

In this paper we introduce the first compressed suffix tree representation that breaks this linear-space barrier. Our representation requires sublinear extra space and supports a large set of navigational operations in logarithmic time. An essential ingredient of our representation is the lowest common ancestor (LCA) query. We reveal important connections between LCA queries and suffix tree navigation.

1 Introduction and Related Work

Suffix trees are extremely important for a large amount of string processing problems. Their many virtues have been described by Apostolico [1] and Gusfield, who dedicates a large portion of his book to them [10]. Once built over a "text" string (which can be done in linear time [24, 16, 22, 4]), suffix trees can find the *occ* occurrences, in the text, of any pattern string, of length m, in the optimal time $O(m + occ)$. Suffix trees also provide elegant linear-time solutions to several complex string problems. A good example is to find the *longest common substring* of two strings, conjectured to have superlinear complexity by Knuth in 1970 [11], and solved in linear time using suffix trees in 1973 [24].

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 [10], 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 has originated a paraphernalia of research results to address it, ranging from space-engineered implementations [9] to novel data structures to simulate it, most notably suffix arrays [14]. 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[3], n being the size of the string and σ the size of the alphabet. For example, the human genome sequence requires 700 Megabytes, while even a space-efficient suffix tree on it requires at least 40 Gigabytes [21], 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 $\log n$.

[3] In this paper log stands for $\log_2$.

These representations are also much larger than the size of the *compressed* string. Recent approaches [17] combining data compression and succinct data structures have achieved spectacular results for the pattern search problem. For example Ferragina *et al.* [5] presented a compressed suffix array that requires $nH_k + o(n \log \sigma)$ bits and computes occ in time $O(m(1 + (\log_\sigma \log n)^{-1}))$. Here nH_k denotes the k-th order empirical entropy of the string [15], a lower bound on the space achieved by any compressor using k-th order modeling.

It turns out that it is possible to use this kind of data structures, that we will call *compressed suffix arrays*[4], and, by adding a few extra structures, support all the operations provided by suffix trees. Sadakane was the first to present such a result [21], adding $6n$ bits to the size of the compressed suffix array. This was later improved to $4n$ extra bits and the same efficiency [6], but the $\Theta(n)$ extra bits space barrier remains.

In this paper we break the $\Theta(n)$ extra-bits space barrier. We build a new suffix tree representation on top of a compressed suffix array, so that we can support all the navigational operations and our extra space fits within the sublinear $o(n \log \sigma)$ extra bits of the compressed suffix array. Our central tool are a particular sampling of suffix tree nodes, its connection with the suffix link and the lowest common ancestor (LCA) query, and the interplay with the compressed suffix array. We exploit the relationship between these actors and in passing uncover some relationships between them that might be of independent interest.

A comparison between Sadakane's representation and ours is shown in Table 2. The result for the time complexities is mixed. Our representation is faster for the important $\textsc{Child}$ operation, yet Sadakane's is usually faster on the rest. On the other hand, our representation requires much less space. For DNA, assuming realistically that $H_k \approx 2$, Sadakane's approach requires $8n + o(n)$ bits, whereas our approach requires only $2n + o(n)$ bits. The recent improvement to $4n + o(n)$ extra bits [6] would still require $6n + o(n)$ bits in this example, three times larger than ours. We choose a compressed suffix array that has the best $\textsc{Letter}$ time, for $nH_k + o(n \log \sigma)$ bits. Only for $O(nH_0) + o(n \log \sigma)$ bits and $\sigma = \omega(\text{polylog}(n))$ is Sadakane's compressed suffix array [20] faster at computing the $\textsc{Letter}$ operation. In that case, using his compressed suffix array, Sadakane's suffix tree would work faster, while ours does not benefit from that. As such, Table 2 shows *the time complexities that can be obtained for suffix trees using the best asymptotic space achieved for compressed suffix arrays alone.* This space is optimal in the sense that no k-th order compressor can achieve asymptotically less space to represent T.

We note that there exists a previous description [7] of a technique based on interval representation and sampling of suffix tree nodes. However the description is extremely brief. Moreover the approach is heuristic and hence no theoretical bounds on the result are given.

2 Basic Concepts

A **string** T is a finite sequence of n symbols taken from a finite **alphabet** Σ of size σ. By $T[i]$ we denote the symbol at position $(i \bmod n)$, where $T[0]$ denotes the first symbol. The **concatenation** of two strings T, T', denoted by $T.T'$, is the string that results from appending T' to T. The **reverse string** T^R of a string T is the string such that $T^R[i] = T[-i-1]$. A **prefix** $T[..i-1]$, **substring** $T[i..j]$ and a **suffix** $T[j+1..]$ of string T are (possibly empty) strings such that $T = T[..i-1].T[i..j].T[j+1..]$.

$\textsc{Parent}(v)$ is the parent node of node v, $\text{F}\textsc{Child}(v)$ gives its first child, and $\text{N}\textsc{Sib}(v)$ the next child of the same parent. $\textsc{Ancestor}(v, v')$ tells whether v is an ancestor of v', that is, if

[4] In the literature these are called compact/compressed suffix arrays, FM-indexes, etc., see [17].

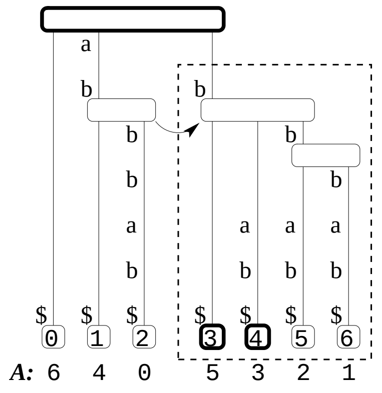

Fig. 1. Suffix tree $\mathcal{T}$ of string *abbbab*, with the leaves numbered. The arrow shows the SLINK between node *ab* and *b*. Bellow it we show the suffix array. The portion of the tree corresponding to node *b* and respective leaves interval is highlighted with a dashed rectangle. The sampled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

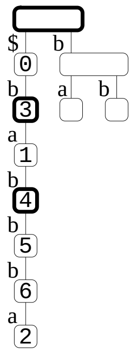

Fig. 2. Reverse tree $\mathcal{T}^R$.

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

   (            (3)(4)        )
i: 0            1 23 4        5
```

Fig. 3. Parentheses representations of trees. The parentheses on top represent the suffix tree and those on the bottom represent the sampled tree. The numbers are not part of the representation; they are shown for clarity. The rows labeled `i:` give the index of the parentheses.

$v = \text{PARENT}^j(v')$ for some $j \geq 0$. $\text{LCA}(v, v')$ is the **lowest common ancestor** of nodes v and v'. $\text{TDEP}(v)$ is the tree-depth of node v, *i.e.* its distance to the ROOT. $\text{LAQT}(v, d)$ (**level-ancestor query**) is the ancestor of v whose tree depth is d.

A **compact tree** is a tree that has no unary nodes, except possibly the ROOT. 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 empty.

The **path-label** of a node v in a labeled tree is the concatenation of the edge-labels from the root down to v. For deterministic trees we refer indifferently to nodes and to their path-labels, also denoted by v. The **string-depth** of a node v in a labeled tree, denoted by $\text{SDEP}(v)$, is the length of its path-label. $\text{LAQS}(v, d)$ is the highest ancestor of node v with $\text{SDEP} \geq d$. $\text{LETTER}(v, i)$ equals $v[i]$, i.e. the i-th letter of the path-label of node v. $\text{CHILD}(v, X)$ is the node that results of descending from node 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 $T\$$, where $\$$ is a terminator symbol not belonging to Σ. We will assume n is the length of $T\$$. The children of suffix tree nodes are lexicographically ordered by the corresponding edge labels. The **suffix-link** of a node $v \neq$ ROOT of a suffix tree, denoted $\text{SLINK}(v)$, is a pointer to node $v[1..]$. Figure 1 shows the suffix tree of string *abbbab*, and illustrates with an arrow the SLINK from the node with path-label *ab* to the node with path-label *b*.[1]

Since the edge-labels that leave from every suffix tree node are lexicographically sorted, the path-labels of the leaves form the set of suffixes of $T\$$ in lexicographical order. Note that $\text{SDEP}(v)$ for a suffix tree leaf v identifies the suffix of $T\$$ starting at position $n - \text{SDEP}(v)$. This operation is called $\text{LOCATE}(v)$. In Figure 1, for leaf $ab\$$ we obtain $T[7-3..] = ab\$$.

A **suffix array** is the array $A[0, n-1]$ of the starting positions of the suffixes of $T\$$ such that the suffixes are lexicographically sorted. In other words, A is the sequence of the $n - \text{SDEP}(v)$ values of the suffix tree leaves, read left to right. Figure 1 shows the suffix array of string *abbbab*. Suffix arrays are able to represent an interesting subset of suffix tree operations [14] by means of

Table 1. Comparing compressed suffix tree representations. The operations are defined along Section 2. Time complexities, but not space, are big-O expressions. Notice that Letter(v, i) can also be computed in $O(i\psi)$ time. Also Child can, alternatively, be computed using FChild and at most σ times NSib. We give the generalized performance and an instantiation using $\delta = (\log_\sigma \log n) \log n$, assuming $\sigma = O(\text{polylog}(n))$, and using the FM-Index of Ferragina *et al.* [5] as the compressed suffix array (CSA).

	Sadakane's		Ours	
Space in bits	$\lvert CSA\rvert + \mathbf{6n} + o(n)$	$= nH_k + \mathbf{6n} + o(n \log \sigma)$	$\lvert CSA\rvert + O((n/\delta) \log n)$	$= nH_k + O((n \log \sigma)/ \log \log n)$ $= nH_k + o(n \log \sigma)$
SDep/Locate	Φ	$= (\log_\sigma \log n) \log n$	$\Psi\delta$	$= (\log_\sigma \log n) \log n$
Count/Ancestor	1	$= 1$	1	$= 1$
Parent	1	$= 1$	$(\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
FChild/NSib	1	$= 1$	$(\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
SLink	Ψ	$= 1$	$(\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
SLink^i	Ψ	$= 1$	$\Phi + (\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
Letter(v, i)	Φ	$= (\log_\sigma \log n) \log n$	Φ	$= (\log_\sigma \log n) \log n$
LCA	1	$= 1$	$(\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
Child	$\Phi \log n$	$= (\log_\sigma \log n)(\log n)^2$	$\log \sigma + \Phi \log \delta + (\Psi + t)\delta$	$= (\log \log n)^2 \log_\sigma n$
TDep	1	$= 1$	$(\Psi + t)\delta^2$	$= ((\log_\sigma \log n) \log n)^2$
LAQt	1	$= 1$	$\log n + (\Psi + t)\delta^2$	$= ((\log_\sigma \log n) \log n)^2$
LAQs	—		$\log n + (\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
WeinerLink	t	$= 1$	t	$= 1$

identifying suffix tree nodes with suffix array intervals: Just as each the i-th suffix tree leaf can be identified with the i-th suffix array entry, internal suffix tree nodes v are identified with the *range* of suffix array entries that correspond to the leaves that descend from v. In Figure 1, the node with path-label b is represented by the interval $[3, 6]$.

We use this customary representation for suffix tree nodes in our proposal. Any suffix tree node v will be represented by the interval $[v_l, v_r]$ of the left-to-right leaf indexes that descend from v. Leaves will also be identified with their left-to-right index (starting at 0) and also with the interval $[v, v]$. For example, if node v is represented as $[v_l, v_r]$, by node $v_l - 1$ we refer to the leaf immediately before v_l, *i.e.* the leaf represented as $[v_l - 1, v_l - 1]$.

An immediate consequence is that we can Count in constant time the number of leaves that descend from any node v. For example, the number of leaves below node b in Figure 1 is $4 = 6 - 3 + 1$. This is precisely the number of times that the string b occurs in the indexed string T, and shows how suffix trees can be used for finding strings in T. This simple application of suffix trees uses only operations Child, Letter, SDep and Count.

Another property of the interval representation is that we solve Ancestor in $O(1)$ time: Ancestor$(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$. This works because suffix trees are compact (*i.e.* no node has just one child) and thus no two different nodes have the same interval.

3 Using Compressed Suffix Arrays

We are interested in compressed suffix arrays because they have very compact representations and support partial suffix tree functionality (being usually more powerful than the classical suffix arrays [17]). Apart from the basic functionality of retrieving $A[i] =$ Locate(i), which we support by means of $n -$ SDep(v) in our representation, state-of-the-art compressed suffix arrays support operation SLink(v) *for leaves* v. This is called $\psi(v)$ in the literature: $A[\psi(v)] = A[v] + 1$, and thus SLink$(v) = \psi(v)$ for a leaf v. We denote by $O(\Psi)$ the time complexity of this operation for a particular compressed index. They also support the WEINERLINK(v, a) operation [24] for nodes v: WEINERLINK(v, X) gives the suffix tree node with path-label $X.v[0..]$. This is called the LF mapping in compressed suffix arrays, and is a kind of inverse of ψ. Consider in Figure 1 the interval $[3, 6]$ that represents a set of 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 denote by $O(t)$ the time to compute this operation on a particular compressed suffix array. For shortness we use an extension of LF to strings, $\mathrm{LF}(X.Y, v) = \mathrm{LF}(X, \mathrm{LF}(Y, v))$, implemented by successively applying LF.

The iterated version of ψ, denoted as ψ^i, can usually be computed faster than $O(i\Psi)$ with compressed indexes. This is achieved with the inverse permutation A^{-1} and LOCATE [17]. We denote its time complexity by $O(\Phi)$.

Finally, 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 $O(\Phi + \ell\Psi)$ 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[5]. This corresponds to $v[0]$, the first letter of the path-label of a suffix tree leaf v.

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 Sadakane's CSA [20], which requires $\frac{1}{\epsilon}nH_0 + O(n \log \log \sigma)$ bits of space and has times $\Psi = O(1)$, $\Phi = O(\log^\epsilon n)$, and $t = O(\log n)$, for any $\epsilon > 0$. For our results we favor a second compressed suffix array, called the FM-index [5], which requires $nH_k + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. Its complexities are $\Psi = t = O(1 + (\log_\sigma \log n)^{-1})$ and $\Phi = (\log_\sigma \log n) \log n$.[6] The instantiation in Table 2 is computed for the FM-index, but the reader can easily compute the result of using Sadakane's CSA. In that case the comparison would favor more Sadakane's compressed suffix tree, yet the space would be considerably higher.

4 The Sampled Suffix Tree

A pointer based implementation of suffix trees requires $O(n \log n)$ bits to represent a suffix tree of (at most) $2n$ 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 $\delta = \omega(\log_\sigma n)$, the sampled tree can be represented using $o(n \log \sigma)$ bits. To fix ideas we can assume $\delta = \lceil (\log_\sigma \log n) \log n \rceil$. In our running example we use $\delta = 4$.

To understand the structure of the sampled tree notice that every tree with $2n$ nodes can be represented in $4n$ bits as a sequence of parentheses (see Figures 1 and 3). The representation of the sampled tree can be 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 1. *A* δ-sampled tree *S of a suffix tree $\mathcal{T}$ with $\Theta(n)$ nodes is formed by choosing $O(n/\delta)$ nodes of $\mathcal{T}$ so that for each node v of $\mathcal{T}$ there is an $i < \delta$ such that node* SLINK$^i(v)$ *is sampled. The* PARENT *of a sampled node in S is the lowest sampled ancestor of its* PARENT *in $\mathcal{T}$.*

[5] This is determined in constant time as the $c \in \Sigma$ satisfying $C[c] \leq i < C[c + 1]$, see Navarro *et al.* [17].

[6] $\psi(i)$ can be computed as $select_{T[A[i]]}(T^{bwt}, T[A[i]])$ using the multiary wavelet tree [12]. The cost for Φ is obtained using a sampling step of $(\log_\sigma \log n) \log n$, so that $o(n \log \sigma)$ stands for $O((n \log \sigma)/ \log \log n)$ as for our other suffix tree structures.

This means that if we start at v and follow suffix links successively, *i.e.* v, $\textsc{SLink}(v)$, $\textsc{SLink}(\textsc{SLink}(v))$, $\ldots$, we will find a sampled node in at most δ steps. Note that this property implies that the $\textsc{Root}$ must be sampled, since $\textsc{SLink}(\textsc{Root})$ is undefined. Appendix A shows that it is feasible to obtain S in linear time from the suffix tree.

In addition to the pointers, that structure the sampled tree, we store in the sampled nodes their interval representation; their $\textsc{SDep}$ and $\textsc{TDep}$; the information to answer lowest common ancestor queries in S, LCA_S, in constant time [2, 6]; and the information to answer level ancestor queries in S, LAQ_S, in constant time [3, 8]. All this requires $O((n/\delta)\log n)$ bits of space, as well as some further data introduced later.

In order to make effective use of the sampled tree, we need a way to map any node $v = [v_l, v_r]$ to its *lowest sampled ancestor*, $\text{LSA}(v)$. A second operation of interest is the *lowest common sampled ancestor* between two nodes:

Definition 2. *Let v, v' be nodes of the suffix tree $\mathcal{T}$. Then,* $\text{LCSA}(v, v')$ *is the lowest common sampled ancestor of v and v', that is, the lowest common ancestor that is represented in the sampled tree S.*

In Figure 1 the $\text{LCSA}(3, 4)$ node is the $\textsc{Root}$, whereas $\text{LCA}(3, 4)$ is $[3, 6]$, *i.e.* the node labeled b.

It is not hard to see that $\text{LCSA}(v, v') = \text{LCA}_S(\text{LSA}(v), \text{LSA}(v')) = \text{LSA}(\text{LCA}(v, v'))$. Next we show how LSA is supported for leaves in constant time and $O((n/\delta)\log n)$ extra bits. With that we also have LCSA in constant time (for leaves, we will extend this, for a general node, later).

4.1 Computing LSA for Leaves

LSA is computed by using an operation $\textsc{Reduce}(v)$, 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 by 5 in Figure 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 $\textsc{Root}$. We assume $\textsc{Reduce}$ 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:

$$\text{LSA}(v) = \begin{cases} \textsc{Reduce}(v) & \text{, if the parenthesis at } \textsc{Reduce}(v) \text{ is '('} \\ \textsc{Parent}(\textsc{Reduce}(v)) & \text{, otherwise} \end{cases}$$

To compute $\textsc{Reduce}$ we use a bitmap $RedB$ and an array $RedA$. The bitmap $RedB$ is initialized with zeros. For every sampled node v represented as $[v_l, v_r]$ we set bits $RedB[v_l]$ and $RedB[v_r + 1]$ to 1. For example the $RedB$ in our example is 1001110. This bitmap indicates the leaves for which we must store partial solutions to $\textsc{Reduce}$. In our example the leaves are $0, 3, 4, 5$. These partial solutions are stored in array $RedA$ (in case of a collision $v_r + 1 = v'_l$, the data for v'_l is stored). In our example these partial results are respectively $0, 1, 3, 4$. Therefore $\textsc{Reduce}(v) = RedA[\textsc{Rank}_1(RedB, v) - 1]$, where v is a leaf number and $\textsc{Rank}_1$ counts the number of 1's in $RedB$ up to and including position v.

First we show that $\textsc{Reduce}$ can be computed in $O(1)$ time with $O((n/\delta)\log n)$ bits. The bitmap $RedB$ cannot be stored in uncompressed form because it would require n bits. We store $RedB$ with the representation of Raman *et al.* [18] that needs only $m \log \frac{n}{m} + o(n)$ bits, where $m = O(n/\delta)$ is the number of 1's in the bitmap (as every sampled node inserts at most two 1's in $RedB$). Hence $RedB$ needs $O((n/\delta)\log\delta) = O((n/\delta)\log n)$ bits, and supports $\textsc{Rank}_1$ in $O(1)$ time. On the other

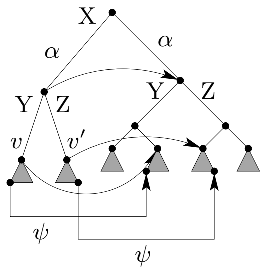

Fig. 4. Schematic representation of the relation between LCA and SLINK, see Lemma 1. Curved arrows represent SLINK and straight arrows the ψ function.

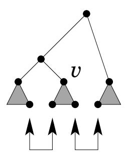

Fig. 5. Schematic PARENT operation. Double arrows indicate the LCA operations.

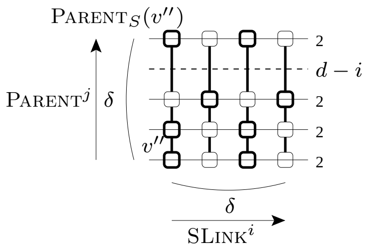

Fig. 6. Schematic representation of the $v_{i,j}$ nodes of the LAQS operation. We represent only the sampling of definition 1 but assume, to simplify the representation, that there are at most δ nodes between v and $\text{PARENT}_S(v'')$

hand, since there are also $O(n/\delta)$ integers in $RedA$, we can store them explicitly to have constant constant access time in $O((n/\delta)\log n)$ bits. Therefore REDUCE can be computed within the assumed bounds. According to our previous explanation, so can LSA and LCSA, for leaves.

5 The Kernel Operations

We have described the two basic components of our compressed suffix tree representation. Most of our functionality builds on the LCA operation, which is hence fundamental to us. In this section we present an entangled mechanism that supports operations LCA, SLINK, and SDEP, depending on each other. The other suffix tree operations are implemented on top of these kernel operations.

5.1 Two Fundamental Observations

We point out that SLINK's and LCA's commute on suffix trees.

Lemma 1. *For any nodes v, v' such that* $\text{LCA}(v, v') \neq \text{ROOT}$ *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 $\text{LCA}(v, v') \neq \text{ROOT}$ is easy to verify, in a suffix tree, by comparing the first letters of the path-label of v and v', *i.e.* $\text{LCA}(v, v') \neq \text{ROOT}$ iff $v[0] = v'[0]$.

Our next lemma shows the fundamental property that relates all the kernel operations.

Lemma 2. *Let v, v' be nodes such that* $\text{SLINK}^r(\text{LCA}(v, v')) = \text{ROOT}$*, 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:

$$\begin{aligned}
\textsc{SDep}(\mathrm{LCA}(v,v')) &= i + \textsc{SDep}(\textsc{SLink}^i(\mathrm{LCA}(v,v'))) & (1)\\
&= i + \textsc{SDep}(\mathrm{LCA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v'))) & (2)\\
&\geq i + \textsc{SDep}(\mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v'))) & (3)
\end{aligned}$$

Equation (1) holds by iterating the fact that $\textsc{SDep}(v'') = 1 + \textsc{SDep}(\textsc{SLink}(v''))$ for any node v'' for which $\textsc{SLink}(v'')$ is defined. Equation (2) results from applying Lemma 1 repeatedly. Inequality (3) comes from Definition 2 and the fact that if node v''' is an ancestor of node v'' then $\textsc{SDep}(v'') \geq \textsc{SDep}(v''')$. Therefore $\textsc{SDep}(\mathrm{LCA}(v,v')) \geq \max_{0 \leq i < d}\{\ldots\}$. On the other hand, from Definition 1 we know that for some $i < \delta$ the node $\textsc{SLink}^i(\mathrm{LCA}(v,v'))$ is sampled. The formula goes only up to d, but $d < \delta$ only if $\textsc{SLink}^d(\mathrm{LCA}(v,v')) = \textsc{Root}$, which is also sampled. According to Definition 2, inequality (3) becomes an equality for that node. Hence $\textsc{SDep}(\mathrm{LCA}(v,v')) \leq \max_{0 \leq i < d}\{\ldots\}$. □

5.2 Entangled Operations

To apply Lemma 2 we need to support operations LCSA, SDep, and SLink. Operation LCSA is supported in constant time, but only for leaves (Section 4.1). Since SDep is applied only to sampled nodes, we have it readily stored in the sampled tree. SLink is also unsolved. We recall from Section 3 that $\textsc{SLink}(v) = \psi(v)$ is supported by the compressed suffix array when v is a leaf. However, we need it for internal tree nodes as well.

Computing SDep. Assuming we can compute SLink, we can use Lemma 2 to compute $\textsc{SDep}(v) = \textsc{SDep}(\mathrm{LCA}(v,v))$, since $\mathrm{LCA}(v,v) = v$.

Computing SLink/SLink^i. Let $v \neq \textsc{Root}$ be represented as $[v_l, v_r]$. One might naively think that $\textsc{SLink}(v)$ could be represented as $[\psi(v_l), \psi(v_r)]$. This however is not always the case: Apply the idea to $\textsc{SLink}(X.\alpha)$ in Figure 4. The correct solution is $\textsc{SLink}(v) = \mathrm{LCA}(\psi(v_l), \psi(v_r))$, as explained by Sadakane [21]. More generally, it holds $\textsc{SLink}^i(v) = \mathrm{LCA}(\psi^i(v_l), \psi^i(v_r))$. Recall that compressed suffix arrays can compute ψ^i efficiently. Thus, we can support SLink provided we have LCA.

Computing LCA The next lemma shows that, if we can compute SDep and SLink, we can compute LCA.

Lemma 3. *Let v, v' be nodes such that $\textsc{SLink}^r(\mathrm{LCA}(v,v')) = \textsc{Root}$ and let $d = \min(\delta, r+1)$. Then there exists an $i < d$ such that*

$$\mathrm{LCA}(v,v') \quad = \quad \mathrm{LF}(v[0..i-1], \mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v')))$$

Proof. This is a direct consequence of Lemma 2. Let i be the index of the maximum of the set in Lemma 2, *i.e.* $\textsc{SLink}^i(\mathrm{LCA}(v,v'))$ is a sampled node and hence it is the same as $\mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v'))$. Note that from the definition of LF mapping we have that $\mathrm{LF}(v''[0], \textsc{SLink}(v'')) = v''$. Applying this iteratively to $\textsc{SLink}^i(\mathrm{LCA}(v,v'))$ 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 $\textsc{SDep}(\mathrm{LCA}(v,v'))$. Accessing the letters to apply LF is not a problem, as we have always to obtain the first letter of a path-label, $\textsc{SLink}^i(v)[0] = \textsc{SLink}^i(v')[0]$. However, we still need SLink and LCSA over internal nodes.

5.3 Breaking the Circle

To get out of this circular dependence situation, we need a new idea. This idea is to handle all the computation over leaves, for which we can compute $\text{SL}\textsc{ink}(v) = \psi(v)$ and $\text{LCSA}(v, v')$.

Lemma 4. *Let v, v' be nodes represented respectively as $[v_l, v_r]$ and $[v'_l, v'_r]$. Then*

$$\text{LCA}(v, v') \quad = \quad \text{LCA}(\min\{v_l, v'_l\}, \max\{v_r, v'_r\})$$

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 $\text{SL}\textsc{ink}$, ψ and the rest of the tree. Using this property and ψ the equation in Lemma 2 reduces to:

$\text{SD}\textsc{ep}(\text{LCA}(v, v'))$

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

Now the circle is broken and $\text{SD}\textsc{ep}(v)$ simplifies to:

$$\text{SD}\textsc{ep}(v) = \text{SD}\textsc{ep}(\text{LCA}(v, v)) = \max_{0 \leq i < d}\{i + \text{SD}\textsc{ep}(\text{LCSA}(\psi^i(v_l), \psi^i(v_r)))\}$$

Operationally, this corresponds to iteratively taking the ψ function of v_l and v_r, δ times or until the $\textsc{Root}$ is reached. At each step we find the LCSA of the two current leaves and retrieve its stored $\text{SD}\textsc{ep}$. The overall process takes $O(\Psi\delta)$ time.

Likewise $\text{LCA}(v, v')$ simplifies to :

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

Now it is finally clear that we do need $\text{SD}\textsc{ep}$ to compute LCA, because we have no other mean to determine which i reaches the sampled node. The arguments to LCSA are not necessarily intervals that correspond to suffix tree nodes. The time to compute LCA is thus $O((\Psi + t)\delta)$. Using LCA we compute $\text{SL}\textsc{ink}$ in $O((\Psi + t)\delta)$ and $\text{SL}\textsc{ink}^i$ in $O(\Phi + (\Psi + t)\delta)$ time.

Finally, note that using Lemma 4 we can extend LSA for a general node v as $\text{LSA}(v) = \text{LSA}(\text{LCA}(v, v)) = \text{LSA}(\text{LCA}(v_l, v_r)) = \text{LCSA}(v_l, v_r)$.

6 Further Operations

We now show how the other operations can be computed on top of the kernel ones, going from simpler to more complex operations.

Computing $\textsc{Letter}$: Since $\textsc{Letter}(v, i) = \text{SL}\textsc{ink}^i(v)[0] = \psi^i(v_l)[0]$, we can solve it in time $O(\min(\Phi, i\Psi))$.

Computing Parent: For any node v represented as $[v_l, v_r]$ we have that $\text{Parent}(v)$ is either $\text{LCA}(v_l - 1, v_l)$ or $\text{LCA}(v_r, v_r + 1)$, whichever is lowest (see Figure 5). This computation is correct because suffix trees are compact. Notice that if one of these nodes is undefined, either because $v_l = 0$ or $v_r = n$, then the parent is the other node. If both nodes are undefined the node in question is the Root which has no Parent node.

Computing Child: Suppose for a moment that every sampled node stores a list of its children and the corresponding first letters of the edges. In our example the Root would store the list $\{(\$, [0, 0]), (a, [1, 2]), (b, [3, 6])\}$, which can reduced to $\{(\$, 0), (a, 1), (b, 3)\}$. Hence, for sampled nodes, it would be possible to compute $\text{Child}(v, X)$ in $O(\log \sigma)$ by binary searching its child list. To compute Child on non-sampled nodes we could use a process similar to Lemma 3: determine which $\text{SLink}^i(v)$, with $i < \delta$, is sampled; compute $\text{Child}(\text{SLink}^i(v), X)$; and use the LF mapping to obtain the answer, *i.e.* $\text{Child}(v, X) = \text{LF}(v[0..i - 1], \text{Child}(\text{SLink}^i(v), X))$. This process requires $O(\log \sigma + (\Psi + t)\delta)$ time. Still, it requires too much space since it may need to store $O(\sigma n/\delta)$ integers.

To avoid exceeding our space bounds we *mark* one leaf out of δ, *i.e.* mark leaf v if $v \equiv_\delta 0$. Do not confuse this concept with sampling, they are orthogonal. In Figure 1 we mark leaves 0 and 4. For every sampled node, instead of storing a list with all the children, we consider only the children that contain marked leaves. In the case of the Root this means excluding the child $[1, 2]$, hence the resulting list is $\{(\$, 0), (b, 3)\}$. A binary search on this list no longer returns only one child. Instead, it returns a range of, at most δ, children. Therefore it is necessary to do a couple of binary searches, inside that range, to delimit the interval of the correct child. This requires $O(\Phi \log \delta)$ time because now we must use Letter to drive the binary searches. Overall, we can compute $\text{Child}(v, X)$ in $O(\log \sigma + \Phi \log \delta + (\Psi + t)\delta)$. Let us now consider space. Ignoring unary paths in the sampled tree, whose space is dominated by the number of sampled nodes, the total number of integers stored amortizes to $O(n/\delta)$, the number of marked leaves. Hence this approach requires at most $O((n/\delta) \log n)$ bits.

Computing TDep: To compute $\text{TDep}(v)$ we need to add other $O(n/\delta)$ nodes to the sampled tree S, so as to guarantee that, for any suffix tree node v, $\text{Parent}^j(v)$ is sampled for some $0 \leq j < \delta$ (Appendix A shows, as a byproduct, how to achieve this too). Recall that the $\text{TDep}(v)$ values are stored in S. Hence, computing $\text{TDep}(v)$ consists in reading $\text{TDep}(\text{LSA}(v))$ and adding the number of nodes between v and $\text{LSA}(v)$, *i.e.* $\text{TDep}(v) = \text{TDep}(\text{LSA}(v)) + j$ when $\text{LSA}(v) = \text{Parent}^j(v)$. The sampling guarantees that $j < \delta$. Hence to determine j we iterate Parent until reaching $\text{LSA}(v)$. The total cost is $O((\Psi + t)\delta^2)$.

Computing LAQt: We extend the $\text{Parent}_S(v)$ notation to represent $\text{LSA}(v)$ when v is a non-sampled node. Recall that the sampled tree supports constant-time level ancestor queries. Hence we have any $\text{Parent}_S^i(v)$ in constant time for any node v and any i. We binary search $\text{Parent}_S^i(v)$ to find the node v' with $\text{TDep}(v') \geq d > \text{TDep}(\text{Parent}_S(v'))$. Notice that this can be computed evaluating only the second inequality. Now we iterate the Parent operation, from v', exactly $\text{TDep}(v') - d$ times. We need the additional sampling introduced for TDep to guarantee $\text{TDep}(v') - d < \delta$. Hence the total time is $O(\log n + (\Psi + t)\delta^2)$.

Computing LAQs: As for LAQt, we start by binary searching $\text{Parent}_S^i(v)$ to find a node v' for which $\text{SDep}(v') \geq d > \text{SDep}(\text{Parent}_S(v'))$. Now we scan all the sampled nodes $v_{i,j} = \text{Parent}_S^j(\text{LSA}(\text{SLink}^i(v')))$ for which $d \geq i + \text{SDep}(v_{i,j})$ and $i, j < \delta$. This means that we start at node v', follow SLink's, 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 $d-(\text{SDep}(v_{i,j})+i) \geq 0$, and then apply the LF mapping to it. The answer is necessarily among the nodes considered.

The time to perform this operation depends on the number of existing $v_{i,j}$ nodes. Appendix A shows how to obtain a sampling that satisfies Definition 1 with at most two distinct sampled nodes SLink$^i(v')$ for any v'. Therefore, there are at most 2δ nodes $v_{i,j}$ (see Figure 6). The additional sampling of SDep adds another 2δ nodes $v_{i,j}$. Hence the total time is $O(\log n + (\Psi + t)\delta)$.

Unfortunately, the same trick does not work for TDep and LAQt, because we cannot know which is the "right" node without bringing all 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 ask for LAQs$(v_l, \text{SDep}(v) + 1)$. Likewise if we use v_r we obtain the last child.

By TDep$_S(v) = i$ we mean that Parent$^i_S(v) =$ Root. This is also defined when v is not sampled. It is possible to skip the binary search step by choosing $v' =$ Parent$^i_S(v_l)$, for $i =$ TDep$_S(v_l)$ $-$ TDep$_S(v)$.

Computing NSib: The next sibling of $v = [v_l, v_r]$ is LAQs$(v_r + 1, \text{SDep}(\text{Parent}(v)) + 1)$ for any $v \neq$ 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.

We are ready to state our summarizing theorem.

Theorem 1. *Using a compressed suffix array (CSA) that supports ψ, ψ^i, $T[A[v]]$ and* LF *in times $O(\Psi)$, $O(\Phi)$, $O(1)$, and $O(t)$, respectively, it is possible to represent a suffix tree with the properties given in Table 2.*

7 Conclusions and Future Work

We presented a fully-compressed representation of suffix trees, which breaks the linear-bits space barrier of previous representations at a reasonable (and in some cases no) 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. In fact our representation is largely based on the LCA operation. Suffix trees have been used in combination with LCA's for a long time, but our results show new ways to explore this partnership.

With respect to practical considerations, we believe that the structure can be implemented without large space costs associated to the sublinear term $o(n \log \sigma)$. In fact, by using parentheses representations of the sampled tree and compressed bitmaps, it seems possible to implement the tree with $\log n + O(\log \delta)$ bits per sampled node. For simplicity we have omitted those improvements. The only datum that seems to need $\log n$ bits is SDep, and even this one should take much less space except in pathological situations.

Our structure has the potential of using much less space than alternative suffix tree representations. On the other hand, we can tune the space/time tradeoff parameter δ to fit the real space needs of the application. Even though some DNA sequences require 700 Megabytes, that is not always the case. Hence it is reasonable to use larger representations of the suffix tree to obtain faster operations, as long as the structure fits in main memory. We believe that our structure should be competitive in practice, using the same amount of space, with alternative implemented schemes [23].

An important challenge is to build this structure in compressed space, as well as supporting a dynamic scenario where the indexed text can be updated. The recent work on dynamic compressed suffix arrays and bitmaps [13] should be relevant for this goal.

Overall we believe this work presents a very significant contribution to the theory of suffix trees. Ours and similar results are eliminating the space handicap normally associated with suffix trees. Thus this type of work has the potential to make suffix trees as popular amongst computer scientists as they should be, given the huge amount of stringology problems they solve.

References

1. A. Apostolico. *The myriad virtues of subword trees*, pages 85–96. Combinatorial Algorithms on Words. NATO ISI Series. Springer-Verlag, 1985.
2. M. Bender, M. Farach-Colton. The LCA problem revisited. In Proceedings of *LATIN*, LNCS 1776, pages 88–94, 2000.
3. M. Bender, M. Farach-Colton. The level ancestor problem simplified. *Theor. Comp. Sci.*, 321(1):5–12, 2004.
4. M. Farach. Optimal suffix tree construction with large alphabets. In Proceedings of *FOCS*, pages 137–143, 1997.
5. P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algor.*, 3(2):article 20, 2007.
6. J. Fischer, V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In Proceedings of *ESCAPE*, LNCS 4614, pages 459–470, 2007.
7. L. Foschini, R. Grossi, A. Gupta, J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algor.*, 2(4):611–639, 2006.
8. R. Geary, R. Raman, V. Raman. Succinct ordinal trees with level-ancestor queries. In Proceedings of *SODA*, pages 1–10, 2004.
9. R. Giegerich, S. Kurtz, J. Stoye. Efficient implementation of lazy suffix trees. *Softw., Pract. Exper.*, 33(11):1035–1049, 2003.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.
11. D. Knuth, J. M. Jr., V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
12. S. Lee, K. Park. Dynamic rank-select structures with applications to run-length encoded texts. In Proceedings of *CPM*, LNCS 4580, pages 96–106, 2007.
13. V. Mäkinen, G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In Proceedings of *CPM*, LNCS 4009, pages 307–318, 2006.
14. U. Manber, E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
15. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
16. E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 32(2):262–272, 1976.
17. G. Navarro, V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
18. R. Raman, V. Raman, S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of *SODA*, pages 233–242, 2002.
19. L. Russo, A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. In Proceedings of *SPIRE*, LNCS 4209, pages 163–180, 2006.
20. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.
21. K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theo. Comp. Sys.*, 2007.
22. E. Ukkonen. Construting suffix trees on-line in linear time. *Algorithmica, 14(3)*, pages 249–260, 1995.
23. N. Välimäki, W. Gerlach, K. Dixit, V. Mäkinen. Engineering a compressed suffix tree implementation. In Proceedings of *WEA*, LNCS 4525, pages 217–228, 2007.
24. P. Weiner. Linear pattern matching algorithms. In Proceedings of *IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

A A Regular Sampling

We show how to obtain a sampling that respects Definition 1. The fundamental insight necessary for this result is to observe that the suffix-links form a labeled trie. Our solution to this problem is based on the work of Russo *et al.* [19].

Definition 3. *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*.*

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.

To guarantee Definition 1 we use the reverse tree. By $\text{HEIGHT}(v^R)$ we refer to the distance between v and its farthest descendant leaf. We sample the nodes for which $\text{TDEP}(v^R) \equiv_{\delta/2} 0$ and $\text{HEIGHT}(v^R) \geq \delta/2$. The second condition is used to guarantee that we do not sample too many nodes, in particular we avoid sampling too close to the leaves. See Figure 2 for an example of this sampling.

Note that in this context 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 $\text{TDEP}(\text{PARENT}^i(v^R)) = i + \text{TDEP}(v^R)$, 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.

To prove the bound on the number of sampled nodes note that any sampled node has at least $\delta/2$ descendants that are not sampled. To be precise this guarantees that we sample at most $\lfloor 4n/\delta \rfloor$ nodes from a suffix tree with $2n$ nodes.

Note that this method, if applied to $\mathcal{T}$ instead of $\mathcal{T}^R$, ensures the condition we needed for TDEP, LAQT and LAQS.