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 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 has originated a plethora of research results, ranging from space-engineered implementations [3] to novel data structures to simulate it, most notably suffix arrays [4]. Some of those space-reduced variants give away some functionality in exchange. For example suffix

*** Supported by the Portuguese Science and Technology Foundation by grants SFRH/BD/12101/2003, SFRH/BPD/34373/2006 and projects DBYeast, POSI/EIA/57398/2004 and ARN, PTDC/EIA/67722/2006.

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

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 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 problem 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 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 [8], 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 [5], adding $6n$ bits to the size of the compressed suffix array.

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 tools 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 uncover some relationships between them that might be of independent interest.

A comparison between Sadakane's representation and ours is shown in Table 1. The result for the time complexities is mixed. Our representation is faster for the important CHILD operation, when $\log \sigma = o(\log \log n)$, 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. We choose a compressed suffix array that has the best LETTER time, for $nH_k + o(n \log \sigma)$ bits. Only when $\sigma = \omega(\mathrm{polylog}(n))$ and there are $O(nH_0) + o(n \log \sigma)$ bits available is Sadakane's compressed suffix array [9] faster at computing the 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 1 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.

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

[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. 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.* [7] as the compressed suffix array (CSA).

	Sadakane's		Ours	
Space in bits	$\lvert CSA\rvert + \mathbf{6n} + o(n)$	$= nH_k + \mathbf{6n} + o(n \log \sigma)$	$\lvert CSA\rvert + O((n/\delta) \log n)$	$= nH_k + o(n \log \sigma)$
SDEP/ LOCATE	Φ	$= (\log_\sigma \log n) \log n$	$\Psi\delta$	$= (\log_\sigma \log n) \log n$
COUNT/ ANCESTOR	1	$= 1$	1	$= 1$
PARENT/ 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	Φ	$= (\log_\sigma \log n) \log n$	$\Phi + (\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
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	$\Phi \log \sigma$	$= (\log \log n) \log n$	$\log \sigma + \Phi \log \delta + (\Psi + t)\delta$	$= (\log \log n)^2 \log_\sigma n$
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	**Not Supported**		$\log n + (\Psi + t)\delta$	$= (\log_\sigma \log n) \log n$
WEINERLINK	t	$= 1$	t	$= 1$

There exists a previous description [10] of a technique based on interval representation and sampling of suffix tree. However it is extremely brief and no theoretical bounds on the result are given.

2 Basic Concepts

Figure 1 shows an example that illustrates the concepts in this section. We denote by T a **string**; by Σ the **alphabet** of size σ; by $T[i]$ the symbol at position $(i \bmod n)$; 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 FCHILD(v) its first child; by NSIB(v) the next child of the same parent; by LAQT(v, d) its **level-d ancestor**; 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. LAQS(v, d) is the highest ancestor of node v with SDEP $\geq d$. 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**

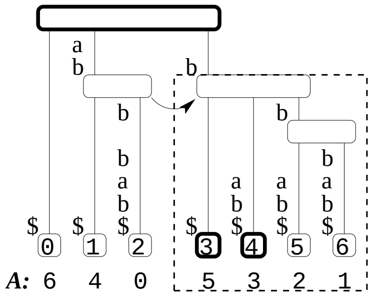

Fig. 1. Suffix tree $\mathcal{T}$ of string *abbbab*, with the leaves numbered. The arrow shows the SLink between node *ab* and *b*. Below it we show the suffix array. The portion of the tree corresponding to node *b* and respective leaves interval is highlighted with a dashed rectangle. The sampled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

```
                  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. 2. 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.

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 $T\$$. 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 $T[$Locate$(ab\$)..] = T[7-3..] = T[4..] = ab\$$. The **suffix array** $A[0, n-1]$ stores the Locate values of the leaves in lexicographicall order. 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: Ancestor$(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$.

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 [6]). Apart from the basic functionality of retrieving $A[i] =$ Locate(i), 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$(v) = \psi(v)$, let its time complexity be $O(\Psi)$. The iterated version of ψ, denoted as ψ^i, can usually be computed faster than $O(i\Psi)$

with compressed indexes [6]. This is achieved with the A and A^{-1}, let its time complexity be $O(\Phi)$. It also supports the WEINERLINK(v, a) operation [11] 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, let its time complexity be $O(t)$. 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 use an extension of LF to strings, $\mathrm{LF}(X.Y, v) = \mathrm{LF}(X, \mathrm{LF}(Y, v))$.

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 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 [9], 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 [7], which requires $nH_k + o(n \log\sigma)$ bits, for any $k \le \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. Its complexities are $\Psi = t = O(1 + (\log_\sigma \log n)^{-1})$ and $\Phi = O((\log_\sigma \log n) \log n)$.[6] The instantiation in Table 1 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 2). The representation of the sampled tree can be obtained by deleting the parentheses of the non-sampled nodes, as in Figure 2. 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.

[5] This is computed in $O(1)$ as the $c \in \Sigma$ satisfying $C[c] \le i < C[c+1]$, see [6].

[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 our other structures.

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* $\text{SLink}^i(v)$ *is sampled.*

This means that if we start at v and follow suffix links successively, *i.e.* v, SLink(v), SLink(SLink(v)), ..., we will find a sampled node in at most δ steps. Note that this property implies that the Root must be sampled, since SLink(Root) is undefined. We sample the nodes v for which $\text{SDep}(v) \equiv_{\delta/2} 0$ and there is a another node v' and a string $|T'| \geq \delta/2$ such that $v' = LF(T', v)$. Notice that this guarantees that from the nodes $LF(T'[..i], v)$, for $-1 \leq i \leq |T'|$, only one is sampled. To be precise this guarantees that we sample at most $\lfloor 4n/\delta \rfloor$ nodes from a suffix tree with $2n$ nodes.

In addition to the pointers, that structure the sampled tree, we store in the sampled nodes their interval representation; their SDep and TDep; the information to answer LCA queries in S, LCA_S, in constant time [13, 14]; and the information to LAQ queries in S, LAQ_S, in constant time [15, 16]; some further data is introduced later. All this requires $O((n/\delta)\log n)$ bits of space.

In order to make effective use of the sampled tree, we need a way to map any node v to its *lowest sampled ancestor*, LSA(v). Another important operation is the *lowest common sampled ancestor* LCSA(v, v'), *i.e.* lowest common ancestor in the sampled tree S. For example LCSA(3, 4) is the Root, whereas LCA(3, 4) is [3, 6], *i.e.* the node labeled b. Note 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; later, we extend this to any node).

4.1 Computing LSA for Leaves

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 by 5 in Figure 2. 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.* 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:

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

To compute 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. In our example $RedB$ is 1001110. This bitmap indicates the leaves for which we must store partial solutions to 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

$\text{Reduce}(v) = RedA[\text{Rank}_1(RedB, v) - 1]$, where v is a leaf number and Rank_1 counts the number of 1's in $RedB$ up to and including position v.

First we show that 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.* [17] 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 Rank_1 in $O(1)$ time. On the other hand, since there are also $O(n/\delta)$ integers in $RedA$, we can store them explicitly to have 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 and SLink depending on each other.

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 3 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 $v[0] = v'[0]$.

The next Lemma shows a fundamental property for 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:

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

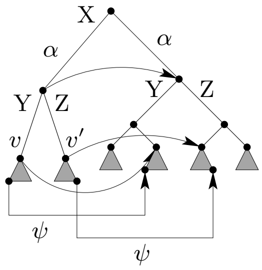

Fig. 3. Schematic representation of the relation between LCA and SLINK, see Lemma 1. Curved arrows represent SLINK and straight arrows the ψ function.

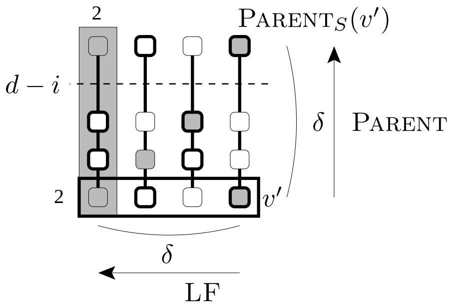

Fig. 4. Schematic representation of the $v_{i,j}$ nodes of the LAQS operation. The nodes sampled because of definition 1 are in bold and the nodes sampled because of the condition of TDEP are filled.

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

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. Sadakane [5] showed that $\text{SLINK}(v) = \text{LCA}(\psi(v_l), \psi(v_r))$, whenever $v \neq \text{ROOT}$. This is not necessarily equal to the $[\psi(v_l), \psi(v_r)]$ interval, see node $X.\alpha$ in Figure 3. In general $\text{SLINK}^i(v) = \text{LCA}(\psi^i(v_l), \psi^i(v_r))$.

Hence all we need is to support LCA. However this depends on Lemma 2.

Lemma 3. $\text{LCA}(v, v') = \text{LF}(v[0..i-1], \text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v')))$ *for any nodes* v, v'*, where* i *is given by Lemma 2.*

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.* $\text{SLINK}^i(\text{LCA}(v, v'))$ is a sampled node and hence it is the same as $\text{LCSA}(\text{SLINK}^i(v), \text{SLINK}^i(v'))$. Note that from the definition of LF mapping we have that $\text{LF}(v''[0], \text{SLINK}(v'')) = v''$. Applying this iteratively to $\text{SLINK}^i(\text{LCA}(v, v'))$ we obtain the equality in the lemma. □

To use this lemma we must know which is the correct i. This is easily determined if we first compute $\text{SDEP}(\text{LCA}(v, v'))$. Accessing the letters to apply LF is not a problem, as we have always to obtain the first letter of a path-label, $\text{SL}\textsc{ink}^i(v)[0] = \text{SL}\textsc{ink}^i(v')[0]$.

5.3 Breaking the Cycle

To get out of this dependency we need a new idea. We will 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. $\text{LCA}(v, v') = \text{LCA}(\min\{v_l, v'_l\}, \max\{v_r, v'_r\})$ *for any nodes* v, v'*.*

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 $v''_l \le v_l, v'_l$ and $v''_r \ge v_r, v'_r$ since v'' is an ancestor of v and v'. This means that $v''_l \le \min\{v_l, v'_l\} \le \max\{v_r, v'_r\} \le v''_r$, *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 $v''_l \le v'''_l \le v'''_r \le v''_r$. Using a similar reasoning for v''' we conclude that $v'''_l \le v''_l \le v''_r \le v'''_r$ and hence $v'' = v'''$. □

Observe this property in Figure 3; ignore $\text{SL}\textsc{ink}$, ψ and the rest of the tree. Using this property and ψ the equation in Lemma 2 reduces to:

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

Operationally, this corresponds to iteratively taking the ψ function, 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 $\text{SD}\textsc{ep}$ and LCA simplify to:

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

Now it is finally clear that we do not need $\text{SL}\textsc{ink}$ to compute LCA. 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. Note that the arguments to LCSA do not correspond necessarily to nodes. Note also 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 other operations can be computed on top of the kernel ones.

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 $\textsc{Parent}$: For any node v represented as $[v_l, v_r]$ we have that $\textsc{Parent}(v)$ is either $\text{LCA}(v_l - 1, v_l)$ or $\text{LCA}(v_r, v_r + 1)$, whichever is lowest. 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 $\textsc{Root}$ which has no $\textsc{Parent}$ 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 be reduced to $\{(\$, 0), (a, 1), (b, 3)\}$. Hence, for sampled nodes, it would be possible to compute $\text{CHILD}(v, X)$ in $O(\log \sigma)$ time 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 $O(\log \sigma + \Phi \log \delta + (\Psi + t)\delta)$ time. 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 $0 \leq j < \delta$. Recall that the $\text{TDEP}(v)$ values are stored in S. Notice that $\text{TDEP}(v) = \text{TDEP}(\text{LSA}(v)) + j$ where $\text{LSA}(v) = \text{PARENT}^j(v)$, hence, computing $\text{TDEP}(v)$ consists in reading $\text{TDEP}(\text{LSA}(v))$ and adding the number of nodes between v and $\text{LSA}(v)$. 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 $\text{PARENT}^i_S(v)$ in constant time for any node v and any i. We binary search $\text{PARENT}^i_S(v)$ 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: We start by binary searching $\text{Parent}_S^i(\text{SLink}^{\delta-1}(v))$ to find a node v' for which $\text{SDep}(v') \geq d - (\delta - 1) > \text{SDep}(\text{Parent}_S(v'))$. Now we scan all the sampled nodes $v_{i,j} = \text{Parent}_S^j(\text{LSA}(\text{LF}(v[i..\delta-1], v')))$ with $\text{SDep}(v_{i,j}) \geq d - i$ and $i, j < \delta$. This means that we start at node v', follow LF, reduce every node found to the sampled tree S and use Parent_S until the SDep of the node drops below $d - i$. Our aim is to find the $v_{i,j}$ that minimizes $\text{SDep}(v_{i,j}) - (d - i) \geq 0$, and then apply the LF mapping to it. The answer is necessarily among the nodes considered.

The time to perform this operation depends on the number of existing $v_{i,j}$ nodes. For this operation the sampling must satisfy Definition 1 and the condition of TDep. Each condition contributes with at most two sampled nodes for every δ nodes. Therefore, there are at most 4δ nodes $v_{i,j}$ (see Figure 4). Unfortunately, the same trick does not work for TDep and LAQt, because we cannot know which is the "right" node without bringing all of them back with LF.

Computing FChild: To find the first child of $v = [v_l, v_r]$, where $v_l \neq v_r$, we simply ask for $\text{LAQs}(v_l, \text{SDep}(v) + 1)$. Likewise if we use v_r we obtain the last child. By $\text{TDep}_S(v) = i$ we mean that $\text{Parent}_S^i(v) = \text{Root}$. This is also defined when v is not sampled. It is possible to skip the binary search step by choosing $v' = \text{Parent}_S^i(v_l)$, for $i = \text{TDep}_S(v_l) - \text{TDep}_S(\text{LSA}(v)) - 1$.

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

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 1.*

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. 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.

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. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. of Algorithms **48**(2) (2003) 294–313
10. Foschini, L., Grossi, R., Gupta, A., Vitter, J.: When indexing equals compression: Experiments with compressing suffix arrays and applications. ACM Trans. Algor. **2**(4) (2006) 611–639
11. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. on Switching and Automata Theory. (1973) 1–11
12. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: CPM. LNCS 4580 (2007) 96–106
13. Bender, M., Farach-Colton, M.: The LCA problem revisited. In: LATIN. LNCS 1776 (2000) 88–94
14. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: ESCAPE. LNCS 4614, Springer (2007) 459–470
15. Bender, M., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comp. Sci. **321**(1) (2004) 5–12
16. Geary, R., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. In Munro, J.I., ed.: SODA, SIAM (2004) 1–10
17. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA. (2002) 233–242