Fast Fully-Compressed Suffix Trees

Gonzalo Navarro *
Department of Computer Science
University of Chile, Chile
gnavarro@dcc.uchile.cl

Luís M. S. Russo †
INESC-ID / Instituto Superior Técnico
Technical University of Lisbon, Portugal
lsr@kdbio.inesc-id.pt

Abstract

We speed up the fully-compressed suffix tree representation (FCST), which is the only one using asymptotically optimal space. Classical representations of suffix trees are fast, but require too much space ($O(n \log n)$ bits for a string of length n over an alphabet of size σ, which is considerably more than the $n \log \sigma$ bits needed to represent the string). Modern compressed suffix tree representations are smaller, getting close to the *compressed* string size, and achieve constant to sublogarithmic time for most operations. However, their space is not fully optimal. An exception is the FCST, which achieves fully optimal space but its times are superlogarithmic. Our contribution significantly accelerates the FCST representation, achieving for many operations log-logarithmic times on typical texts. The resulting FCST variant becomes very attractive in terms of space and time, and a promising alternative in practice.

1 Introduction and Related Work

Suffix trees are extremely important for a large number of string processing problems [2]. Their combinatorial properties have a profound impact in the *bioinformatics* field, which needs to analyze large strings of DNA and proteins with no predefined boundaries [7]. This partnership has produced several key 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.

*Funded by Fondecyt grant 1-110066, Chile.

†Funded by FCT, under projects PEst-OE/EEI/LA0021/2013, NetDyn PTDC/EIA-EIA/118533/2010 and DataStorm EXCL/EEI-ESS/0257/2012.

The space problem is so important that it has originated a plethora of research results, ranging from space-engineered implementations [6] to novel data structures to simulate it, most notably suffix arrays [8]. 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[1], where n is the string length and σ the alphabet size. For example the human genome requires 700 Megabytes, while even a space-efficient suffix tree on it requires at least 40 Gigabytes [13], and the reduced-functionality suffix array requires 12 Gigabytes. This problem is particularly evident in DNA because $\log \sigma = 2$ is much smaller than $\log n$.

Those representations are also much larger than the size of the *compressed* string. Recent approaches [11] combining data compression and succinct data structures have achieved spectacular compression on suffix arrays. For example Ferragina et al. [3] presented a compressed suffix array that requires $nH_k + o(n \log \sigma)$ bits and computes *occ* in time $O(m(1+\frac{\log \sigma}{\log \log n}))$, where m is the size of the pattern, *occ* the number of its occurrences in the string, and nH_k the k-th order empirical entropy of the string [9], 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*[2], and, by adding a few extra structures, support all the operations provided by suffix trees. Sadakane was the first to present such a *compressed suffix tree* (CST) [13], adding $6n$ bits to the size of the compressed suffix array. This $\Theta(n)$ extra-bits space barrier was later removed by the fully-compressed suffix tree (FCST) representation [12], and by the entropy-bounded compressed suffix tree (EBCST) [5, 4].

Table 1 clearly shows that FCSTs are the only ones achieving asymptotically optimal space, but also that they should be fairly slow. This is not a consequence of pessimistic bounds: A recent experimental comparison [1] has shown that FCSTs indeed use much less space, but also are an order of magnitude slower than other CSTs. Indeed, they compete only for the CHILD operation or on full traversals, that is, precisely when the other modern representations also require a lot of time. The FCST representation is paying a steep price for its tiny space requirements.

In this paper we propose faster algorithms for navigating the FCST representation, and a modified FCST representation to support it. The last column of Table 1 shows that, on typical texts, our new FCST requires polylog-log time for most operations, which makes it an excellent candidate for a practical alternative dominating most of the space/time tradeoff map.

We start by recalling the basic FCST we build on [12]. Then we show how to speed up operation LCA in Section 3, and operation SDEP in Section 4. Many others are then speeded up as a consequence, or with analogous means, in Section 5.

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

[2] These are also called compact suffix arrays, FM-indexes, etc. [11].

Table 1: Comparison between compressed suffix tree representations. The '=' sign in the last column indicates that the time is the same of the previous column. The instantiation we show assumes $\sigma = O(\text{polylog}(n))$, and uses the CSA of Grossi et al. for CST and EBCST, and the FM-index of Ferragina et al. for FCST. The space given holds for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. The $o(n)$ space term in this instantiation is $O(n/\log\log n)$. The times in the last column hold for typical texts, where $\text{SDEP} = O(\log n)$.

	CST [13]	EBCST [5, 4]	FCST [12]	New FCST
Space in bits	$(1+\frac{1}{\epsilon})nH_k$ $+6n + o(n)$	$(1+\frac{1}{\epsilon})nH_k+o(n)$	$nH_k + o(n)$	$nH_k + o(n)$
ROOT	$O(1)$	$O(1)$	$O(1)$	=
COUNT	$O(1)$	$O(1)$	$O(1)$	=
ANCESTOR	$O(1)$	$O(1)$	$O(1)$	=
PARENT	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O(\log\log n)$
FCHILD	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O((\log\log n)^2)$
NSIB	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O((\log\log n)^2)$
LCA	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O(\log\log n)$
TDEP	$O(1)$	*No support*	$O(\log n \log\log n)$	$O((\log\log n)^3)$
TLAQ	$O(1)$	*No support*	$O(\log n \log\log n)$	$O((\log\log n)^3)$
LETTER(v, i, ℓ)	$O(\ell + \log^\epsilon n)$	$O(\ell + \log^\epsilon n)$	$O(\ell + \log n \log\log n)$	=
CHILD	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n(\log\log n)^2)$	=
LOCATE	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	=
SLINK	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O(\log\log n)$
SLINK^i	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	=
WEINERLINK	$O(\log n)$	$O(\log n)$	$O(1)$	=
SDEP	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$	$O((\log\log n)^2)$
SLAQ	*No support*	$O(\log^{1+\epsilon} n)$	$O(\log n \log\log n)$	$O((\log\log n)^2)$

2 Basic Concepts

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 TLAQ(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. SLAQ(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** of T is the 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. Moreover no two children out of a node share the initial letter. We will assume n is the length of $T\$$. For a detailed explanation see Gusfield's book [7]. The **suffix-link** of a node $v \neq \textsc{Root}$ of a suffix tree, denoted $\textsc{SLink}(v)$, is a pointer to node $v[1..]$, and we denote $\textsc{SLink}^i(v)$ the iterated suffix link operation from node v. The **Weiner link**, denoted $\textsc{WeinerLink}(a, v)$, is a pointer to node $a.v$, if it exists. Note that $\textsc{SDep}(v)$ of a leaf v identifies the suffix of $T\$$ starting at position $n - \textsc{SDep}(v) = \textsc{Locate}(v)$. The **suffix array** $A[0, n-1]$ stores the $\textsc{Locate}$ values of the leaves in lexicographical order. The *suffix tree nodes can be identified with suffix array intervals*: each node corresponds to the *range* of leaves that descend from v. 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 $\textsc{Count}$ in constant time the number of leaves that descend from v. We can also compute $\textsc{Ancestor}$ in $O(1)$ time: $\textsc{Ancestor}(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$. The $\textsc{Rank}_a(T, i)$ operation over a string T counts the number of times that the letter a occurs in T up to position i. Likewise the $\textsc{Select}_a(T, i)$ operation gives the position of the i-th occurrence of a in T.

A **compressed suffix array** (CSA) supports access to any suffix array cell, $A[i]$, and to the inverse permutation, $A^{-1}[i]$. It also implements operations $\Psi(i) = A^{-1}[A[i] + 1]$ in time $O(\psi)$ and $\text{LF}(a, [l, r]) = \textsc{WeinerLink}(a, [v_l, v_r])$ in time $O(\tau)$. The actual values of ψ and τ depend on the underlying CSA. We generalize the LF notation to a string s; the expression $\text{LF}(s, [l, r])$ represents the iteration of LF by using the letters of s from right to left.

The FCST [12] implements the operations in Table 1 by using a CSA and $o(n)$ bits on top of it. Those $o(n)$ bits include a parentheses representation (e.g., [14]) of a **sampled suffix tree**. The sampling contains $O(n/\delta)$ nodes, for some parameter δ, and ensures that, if we take successive $\textsc{SLink}$ steps from any node v, we will find a sampled node before δ iterations. The reason why such a sampling is useful is that suffix trees are self-similar, as explained by the following lemma, where LCSA stands for the LCA operation restricted to the sampled tree.

Lemma 1 ([12]) *Consider nodes v, v' such that $\textsc{SLink}^r(\text{LCA}(v, v')) = \textsc{Root}$ and $d = \min(\delta, r + 1)$. Then*

$$\textsc{SDep}(\text{LCA}(v, v')) \;=\; \max_{0 \leq i < d} \{i + \textsc{SDep}(\text{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v')))\}.$$

They use this relation to solve the kernel operations LCA, $\textsc{SDep}$, and $\textsc{SLink}$, and then derive the others from those.

3 Faster Lowest Common Ancestors

In this section we speed up operation LCA. We start from Lemma 1, which is more an observation about the structure of suffix trees than an operational scheme. A

detailed discussion about how to actually use the lemma is given by Russo et al. [12]. We now explain how to adapt it to compute LCA without computing SDep. The following lemma is useful for this purpose.

Lemma 2 *Let v, v' be nodes such that* $\text{SLink}^r(\text{LCA}(v, v')) = \text{Root}$. *Then there is an i with $0 \leq i < d = \min(\delta, r + 1)$ such that*

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

Proof. According to Lemma 1, there is an i in the interval such that $\text{SDep}(\text{LCA}(v, v'))$ $= i+\text{SDep}(\text{LCSA}(\text{SLink}^i(v), \text{SLink}^i(v')))$. Let $w = \text{LCSA}(\text{SLink}^i(v), \text{SLink}^i(v'))$ and $u = \text{SLink}^i(\text{LCA}(v, v')) = \text{LCA}(\text{SLink}^i(v), \text{SLink}^i(v'))$. Thus w is an ancestor of u, and its string depth is $\text{SDep}(w) = \text{SDep}(\text{LCA}(v, v')) - i = \text{SDep}(u)$, therefore we conclude that $u = w$. Now, by definition, $\text{LF}(v[0..i-1], w) = \text{LF}(v[0..i-1], u) =$ $\text{LCA}(v, v')$. □

The lemma can be used as follows to obtain $\text{LCA}(v, v')$ without computing any SDep information. Let us define the sequence of nodes $u_i = \text{SLink}^i(\text{LCA}(v, v'))$ (which we will not compute) and $w_i = \text{LCSA}(\text{SLink}^i(v), \text{SLink}^i(v'))$. We will compute the w_i nodes for all i, and retain the lowest one in the SLink^i path, by computing $w'_{d-1} = w_{d-1}$ and, for $i = d-2 \ldots 0$, w'_i as the lowest between w_i and $\text{LF}(v[i], w'_{i+1})$ (since these two nodes are one an ancestor of the other, we can use just Ancestor to determine w'_i). By the proof above, at least one w_i will be equal to u_i, and we will spot it in $w'_i = w_i$ and bring it back to w'_0, thus at the end we will have $\text{LCA}(v, v') = w'_0$.

This procedure is essentially what was computed in the original FCST approach, but now we use the Ancestor operation instead of comparing the numerical SDep values. There may seem to be little or no gain from this change: the time to compute LCA operation is still $O((\psi + \tau)\delta)$. However, since we do not need to store SDep information, we can reduce the δ value to $O(\log \log n)$ and still use $O((n \log \log \log n)/ \log \log n) = o(n)$ extra bits of space, hence speeding up the LCA operation. This space bound is determined by the underlying compressed bitmap used to map intervals to the parenthesis tree.

4 Faster String Depth

In this section we explain how to compute SDep in $O(\psi \log \text{SDep}(v) \log \log n)$ time. The idea is to store the SDep information in sampled nodes, using $\lfloor 1 + \log \text{SDep}(v) \rfloor$ bits instead of $\log n$ bits. Our current sampled tree uses parameter $\delta = \lceil \log \log n \rceil$ and requires $o(n)$ bits, but it includes no SDep information. We store this information in another sampled tree, with a sampling that is larger and varies with the string depth of the nodes.

Our basic sampling technique, with parameter δ, is to sample any node v such that $\text{SDep}(v)$ is a multiple of $\delta/2$ and there is another node v' such that $v = \text{SLink}^{\delta/2}(v')$. This guarantees that there are $O(n/\delta)$ sampled nodes and that any other node is at most δ suffix links away from a sampled node [12].

ℓ_v	SDEP(v)							
1	1							
2	3	*2*						
3	7	*6*	5	4				
4	15	14	13	*12*	11	10	9	**8**
5	23	22	21	**20**	19	18	17	16
	31	*30*	29	28	27	26	**25**	24

Figure 1: Schematic representation of the SDEP(v) values that yield sampled nodes, according the band sampling criterion. On the left, the table contains the ℓ_v values. On the right we show the SDEP(v) values that correspond to the same ℓ_v value. We assume $(\delta/2) = 1$ and that there is a chain of suffix links that starts with SDEP 30 and passes through all the values to SDEP 0. The numbers in **bold** correspond to sampled nodes, according to the band criterion; the numbers in *italics* correspond to nodes that are not sampled because they fail the second condition.

For the second sampling, we will divide the tree into bands where the logarithm of SDEP does not change, that is, node v will belong to band $\ell_v = \lfloor 1 + \log \text{SDEP}(v) \rfloor$. The sampling rule will be that v is sampled iff SDEP(v) is a multiple of $(\delta/2)\ell_v$ and there is another node v' such that $\ell_v = \ell_{v'}$ and $v = \text{SLINK}^{(\delta/2)\ell_v}(v')$. An illustration of this rule is shown in Figure 1.

This guarantees that we sample at most one out of $(\delta/2)\ell$ nodes at each band ℓ. Thus, if we spend $O(\log \text{SDEP}(v))$ bits to store SDEP(v) for sampled nodes v, we spend $O(1/\delta)$ bits per node of any level, maintaining the total space in $O(n/\delta)$ bits. The underlying bitmap that identifies those sampled nodes is sparser than the original one that uses sampling value δ for all the bands, and thus it adds up to $O(n \log \delta/\delta) = o(n)$ bits overall.

On the other hand, the maximum distance that we may traverse from a node v of band ℓ occurs when we change band after traversing $\ell\delta - 1$ suffix links (just at the point where the node would have been sampled if it belonged to band ℓ), and then we traverse other $(\ell-1)\delta$ suffix links in band $\ell - 1$. We cannot change band again before carrying out $2^{\ell-2}$ suffix links, so if $2^{\ell-2} > \delta\ell$, we are sure to complete the process in band $\ell - 1$. If ℓ is small, however, it might be that $2^{\ell-2} \leq \delta\ell$ and we change band again without finding a sampled node. But in this case the whole number of suffix links towards the root is at most $2^{\ell-2} + 2^{\ell-3} + \ldots + 2^1 \leq 2^{\ell-1} \leq 2\delta\ell$ anyway. Therefore we carry out $O(\delta\ell) = O(\delta \log \text{SDEP}(v))$ steps to compute SDEP(v). For example, in Figure 1 no node is sampled for the band $\ell_v = 2$.

Finally, we note that now our vector of sampled SDEP values requires a variable number of bits per value. By concatenating, say, the binary representation of the values, omitting initial 0's, we need $O(\log \text{SDEP})$ bits to encode each value SDEP. A further bitmap of the same total length of the concatenation, marking the beginnings of the numbers, and with sublinear-sized structures to support SELECT operations in constant time [10], gives constant-time access within the desired space bounds.

5 Speeding up other Operations

Operation SLAQ. Russo et al. [12] solve operation SLAQ(v) in a sophisticated way, which involves a binary search among the ancestors of a node in the sampled tree (which was said to cost $O(\log n)$ but it can be refined to $O(\log \text{SDEP}(v))$, the computation of SDEP on $O(\delta)$ sampled tree nodes, plus the traversal of $O(\delta)$ nodes with operations Ψ and LF. Since we need to know SDEP on the sampled nodes, we carry out this process on the tree with varying sampling we have described in Section 4, therefore we have $\delta = O(\log \text{SDEP}(v) \log\log n)$. Therefore, the cost of this operation becomes $O((\psi + \tau) \log \text{SDEP}(v) \log\log n)$ time on our new structures.

Basic navigation. Operations PARENT and SLINK are solved with a constant number of LCA operations [12], and therefore can now be solved in time $O((\psi + \tau) \log\log n)$. Similarly, operations FCHILD and NSIB are solved with a constant number of applications of operations SLAQ, SDEP, and PARENT; therefore these operations require now time $O((\psi + \tau) \log \text{SDEP}(v) \log\log n)$.

Tree depth and operation TLAQ. Operation TDEP(v) is solved in the original FCST with a different sampling of the suffix tree, which guarantees that if we take PARENT successively from a node v, we will find a sampled node before δ iterations. The sampled nodes store their tree depths in the original suffix tree. Thus we perform $i < \delta$ PARENT operations until reaching a marked node v', and the answer is $\text{TDEP}(v') + j$. The time is dominated by the $O(\delta)$ PARENT operations.

Since we have to store tree depths in this sampled tree, we use a varying sampling analogous to the one used to store the SDEP information. Here we sample nodes whose depth within the band ℓ is a multiple of $(\delta/2)\ell$ and have a descendant in the same band at distance $(\delta/2)\ell$. By the same arguments as in Section 4, the sampled nodes require $o(n)$ bits, and the number of PARENT operations to carry out is $O(\log \text{TDEP}(v) \log\log n) = O(\log \text{SDEP}(v) \log\log n)$. By multiplying this by the $O((\psi+\tau) \log\log n)$ time used by PARENT (which is carried out in the denser sampled tree), we have a final cost of $O((\psi + \tau) \log \text{SDEP}(v)(\log\log n)^2)$ time.

For TLAQ(v), we find the least sampled ancestor of v in this tree and then binary search for the two sampled nodes whose (stored) TDEP values contain the target depth. Then we sequentially find the right node with successive PARENT operations from the lower node enclosing the target value. The time is the same as for TDEP(v).

6 Discussion and Conclusions

In the previous sections we described how to speed up the FCST representation. Namely we improved the worst-case performance of the LCA, SLINK and PARENT operations to just $O(\log\log n)$ steps, by detaching their computation from that of string depths, SDEP, and thus freeing the FCST from the need to store string depth information at sampled nodes. Thus the sampling can be made much denser, boosting the operations within the same asymptotic space.

The operations still involve the computation of string depths or similar values. We showed how to store $\mathrm{SDEP}(v)$ within $O(\log \mathrm{SDEP}(v))$ bits, which enabled us to compute it in $O(\log \mathrm{SDEP}(v) \log\log n)$ steps, still within $o(n)$ bits of space. This is to be compared to the $O(\log n \log\log n)$ time of the original FCST. The other operations are speeded up analogously.

How significant is the speedup from $O(\log n)$ to $O(\log \mathrm{SDEP}(v))$ depends on how large is $\mathrm{SDEP}(v)$. Szpankowski [15] shows that, for "typical" texts, the maximum string depth is at most $c \cdot \log n$ for some constant c "almost surely" [15, Thm. 1(ii) and Remark 2(iv)]. His definition of typical texts is texts sampled from a stationary mixing ergodic source (more precisely, type A2 in his categorization), a quite general assumption including Bernoulli and Markovian models. The "almost surely" (a.s.) term is a very strong convergence, stronger than "on average", "with high probability", and "infinitely often".[3] Therefore, on typical texts, we have that $\log \mathrm{SDEP}(v) \leq \log\log n + O(1)$ and, for example, $\mathrm{SDEP}(v)$ is computed in $O((\log\log n)^2)$ steps and $\mathrm{TDEP}(v)$ in $O((\log\log n)^3)$ steps.

Theorem 3 *A suffix tree of a text of size n can be represented in $nH_k + o(n \log \sigma)$ bits, which includes the space to store a CSA that implements operations Ψ in $O(\psi)$ time,* LF *in $O(\tau)$ time, and access to the suffix array and its inverse in within time $O(\min(\psi, \tau) \log n \log\log n)$. Then the suffix tree carries out the operations listed in Table 1, in the time bounds given in the last column multiplied by $O(\psi + \tau)$, almost surely on typical texts.*

There are other scenarios, especially in Computational Biology, that are not well described by the ergodic source models implied in "typical" texts. For example, in a collection of genomes of individuals of the same species, we might have that each new genome is almost identical to a previous one, with a mutation rate of p, where p is typically in the range 10^{-2} to 10^{-4}. If a substring of the current genome of length ℓ is equal to a previous one, we will have $\mathrm{SDEP}(v) = \ell$ for the corresponding suffix tree node. The average value of ℓ is $1/p$, and if we regard the distribution as geometric, we expect that the longest match between any two strings be $\frac{2\ln n + O(1)}{\ln \frac{1}{1-p}}$. In this case we expect $\log \mathrm{SDEP}$ to be bounded by $\log\log n - \log\log \frac{1}{1-p} + O(1)$. When p is close to zero, this is approximated by $\log\log n + \log \frac{1}{p} + O(1)$. This adds a nonnegligible, yet still manageable, constant for the typical values of p. A similar case occurs when the sequences are built from chunks extracted at arbitrary positions from a few base sequences.

Our results are a significant improvement over the FCSTs representation, which is currently the smallest (yet slowest) compressed suffix tree representation, both in theory and in practice. The speedups are significant and can potentially bring the FCST representation much closer to the time performance of the other compressed representations. Future work is to implementing these new techniques, making commonsense engineering decisions where necessary.

[3] A sequence X_n tends to a value β almost surely if, for every $\epsilon > 0$, the probability that $|X_N/\beta - 1| > \epsilon$ for some $N > n$ tends to zero as n tends to infinity, $\lim_{n\to\infty} \sup_{N>n} \Pr(|X_N/\beta - 1| > \epsilon) = 0$.

Acknowledgments

We are thankful to Djamal Belazzougui for useful comments.

References

[1] A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[2] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[3] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[4] J. Fischer. Wee LCP. *Information Processing Letters*, 110:317–320, 2010.

[5] J. Fischer, V. Mäkinen, and G. Navarro. Faster Entropy-Bounded Compressed Suffix Trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[6] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software Practice and Experience*, 33(11):1035–1049, 2003.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[8] U. Manber and E. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[9] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[10] I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.

[11] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[12] L. S. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.

[13] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[14] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[15] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.