An(other) Entropy-Bounded Compressed Suffix Tree

Johannes Fischer[1], Veli Mäkinen[2], and Gonzalo Navarro[1]

[1] Dept. of Computer Science, Univ. of Chile. {jfischer|gnavarro}@dcc.uchile.cl
[2] Dept. of Computer Science, Univ. of Helsinki, Finland. vmakinen@cs.helsinki.fi

Abstract. Suffix trees are one of the most important data structures in stringology, with myriads of applications in fluorishing areas like bioinformatics. As their main problem is space usage, recent efforts have focused on compressed suffix tree representations, which obtain large space reductions in exchange for moderate slowdowns. Such a smaller suffix tree could fit in a faster memory, outweighting by far the theoretical slowdown. We present a novel compressed suffix tree. Compared to the current compressed suffix trees, it is the first achieving at the same time sublogarithmic complexity for the operations, and space usage which goes to zero as the entropy of the text does. Our development contains several novel ideas, such as compressing the longest common prefix information, and totally getting rid of the suffix tree topology, expressing all the suffix tree operations using range minimum queries and a new primitive called next/previous smaller value in a sequence.

1 Introduction

Suffix trees are probably the most important structure ever invented in stringology. They have been said to have "myriads of virtues" [2], and also have myriads of applications in many areas, most prominently bioinformatics [12]. One of the main drawbacks of suffix trees is their considerable space requirement, which is usually close to $20n$ bytes for a sequence of n symbols, and at the very least $10n$ bytes [16]. For example, the Human genome, containing approximately 3 billion bases, could easily fit in the main memory of a desktop computer (as each DNA symbol needs just 2 bits). However, its suffix tree would require 30GB to 60GB, too large to fit in normal main memories. Although there has been some progress in managing suffix trees in secondary storage [14] and it is an active area of research [15], it will always be faster to operate the suffix tree in main memory.

This situation has stimulated research on compressed representation of suffix trees, where the data structure operates without uncompressing it. Even if many more operations are needed to carry out the operations on the compressed representation, this is clearly advantageous compared to having to manage it on secondary memory. A large body of research focuses on compressed suffix arrays [21], which offer a reduced suffix tree functionality. In particular, they miss the important suffix-link operation. The same restrictions apply to early compressed suffix trees [20, 11].

The first fully-functional compressed suffix tree is due to Sadakane [25]. It builds on top of a compressed suffix array [24] that uses $\frac{1}{\epsilon}nH_0 + O(n \log\log \sigma)$ bits of space, where H_0 is the zero-order entropy of the text $T_{1,n}$, σ is the size of the alphabet of T, and $0 < \epsilon < 1$ is any constant. In addition, the compressed suffix tree needs $6n + o(n)$ bits of space. Most of the suffix tree operations can be carried out in constant time, except for knowing the string-depth of a node and the string content of an edge, which take $O(\log^\epsilon n)$ time, and moving to a child, which costs $O(\log^\epsilon n \log \sigma)$. One could replace the compressed suffix array they use by Grossi et al.'s [10], which requires less space: $\frac{1}{\epsilon}nH_k + o(n \log \sigma)$ bits for any $k \leq \alpha \log_\sigma n$, where H_k is the k-th empirical entropy of T [18] and $0 < \alpha < 1$ is any constant. However, the $O(\log^\epsilon n)$ time complexities become $O(\log_\sigma^{\frac{\epsilon}{1-\epsilon}} n \log \sigma)$ [10, Thm. 4.1]. In addition, the extra $6n$ bits in the space complexity remain, despite any reduction we can achieve in the compressed suffix array. This term can be split into $2n$ bits to represent

(with a bitmap called *Hgt*) the longest common prefix (LCP) information, plus $4n$ bits to represent the suffix tree topology with parentheses. Many operations are solved via constant-time range minimum queries (RMQs) over the depths in the parentheses sequence. An RMQ from i to $j \geq i$ over a sequence $S[1,n]$ of numbers asks for $\text{RMQ}_S(i,j) := \text{argmin}_{i \leq \ell \leq j} S[\ell]$.

Russo et al. [23] recently achieved fully-compressed suffix trees, that is, requiring $nH_k + o(n \log \sigma)$ bits of space (with the same limits on k as before), which is essentially the space required by the smallest compressed suffix array, and asymptotically optimal under the k-th entropy model. The main idea is to sample some suffix tree nodes and use the compressed suffix array as a tool to find nearby sampled nodes. The most adequate compressed suffix array for this task is the alphabet-friendly FM-index [5]. The time complexities for most operations are logarithmic at best, more precisely, between $O(\log n)$ and $O(\log n \log\log n)$. Others are slightly more expensive, for example moving to a child costs an additional $O(\log\log n)$ factor, and some less common operations are as costly as $O((\log n \log\log n)^2)$.

We present a new fully-compressed suffix tree, by removing the $6n$ term in Sadakane's space complexity. The space we achieve is not as good as that of Russo et al., but most of our time complexities are sublogarithmic. More precisely, our index needs $nH_k(\frac{1}{\epsilon} + 2\log\frac{1}{H_k}) + o(n\log\sigma)$ bits of space. Note that, although this is not the ideal nH_k, it still goes to zero as $H_k \to 0$, unlike the uncompressible $6n$ bits in Sadakane's structure. Our solution builds on two novel algorithmic ideas to improve Sadakane's compressed suffix tree.

1. We show that array *Hgt*, which encodes LCP information in $2n$ bits [25], actually contains $2R$ runs, where R is the number of *runs in* ψ [21]. We show how to run-length compress *Hgt* into $2R\log\frac{n}{R} + o(n)$ bits while retaining constant-time access. In order to relate R with nH_k, we use the result $R \leq nH_k + \sigma^k$ for any k [17], although sometimes it is extremely pessimistic (in particular it is useful only for $H_k < 1$, as obviously $R \leq n$). This gives the $2nH_k\log\frac{1}{H_k}$ upper bound to store *Hgt*.
2. We get rid of the suffix tree topology and identify suffix tree nodes with suffix array intervals. All the tree traversal operations are simulated with RMQs on LCP (represented with *Hgt*), plus a new type of queries called "Next/Previous Smaller Value", that is, given a sequence of numbers $S[1,n]$, find the first cell in S following/preceding i whose value is smaller than $S[i]$.[1] We show how to solve these queries in sublogarithmic time while spending only $o(n)$ extra bits of space on top of S. We believe this operation might have independent interest, and the challenge of achieving constant time with sublinear space remains open.

2 Basic Concepts

The *suffix tree* $\mathcal{S}$ of a text $T_{1,n}$ over an alphabet Σ of size σ is a compact trie storing all the suffixes $T_{i,n}$ where the leaves point to the corresponding i values [2, 12]. For technical convenience we assume that T is terminated with a special symbol, so that all lexicographical comparisons are well defined. For a node v in $\mathcal{S}$, $\pi(v)$ denotes the string obtained by reading the edge-labels when walking from the root to v (the *path-label* of v [23]). The *string-depth* of v is the length of $\pi(v)$.

Definition 1. *A* suffix tree representation *supports the following operations:*

– Root()*: the root of the suffix tree.*

[1] Computing NSVs/PSVs on the fly has been considered in parallel computing [3], yet not in the static scenario.

- LOCATE(v): *the suffix position i if v is the leaf of suffix $T_{i,n}$, otherwise* NULL.
- ANCESTOR(v, w): *true if v is an ancestor of w.*
- SDEPTH(v)/TDEPTH(v): *the string-depth/tree-depth of v.*
- COUNT(v): *the number of leaves in the subtree rooted at v.*
- PARENT(v): *the parent node of v.*
- FCHILD(v)/NSIBLING(v): *the alphabetically first child/next sibling of v.*
- SLINK(v): *the* suffix-link *of v; i.e., the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma$.*
- SLINK^i(v): *the* iterated suffix-link *of v; (node w s.th. $\pi(w) = \beta$ if $\pi(v) = \alpha\beta$ for $\alpha \in \Sigma^i$).*
- LCA(v, w): *the* lowest common ancestor *of v and w.*
- CHILD(v, a): *the node w s.th. the first letter on edge (v, w) is $a \in \Sigma$.*
- LETTER(v, i): *the ith letter of v's path-label, $\pi(v)[i]$.*
- LAQS(v, d)/LAQT(v, d): *the highest ancestor of v with string-depth/tree-depth $\geq d$.*

Existing compressed suffix tree representations include a *compressed full-text index* [21, 24, 10, 5], which encodes in some form the *suffix array* $\mathsf{SA}[1, n]$ of T, with access time t_{SA}. Array SA is a permutation of $[1, n]$ storing the pointers to the suffixes of T (i.e., the LOCATE values of the leaves of $\mathcal{S}$) in lexicographic order. Most full-text indexes also support access to permutation SA^{-1} in time $O(t_{\mathsf{SA}})$, as well as the efficient computation of permutation $\psi[1, n]$, where $\psi(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] + 1]$ for $1 \leq i \leq n$ if $\mathsf{SA}[i] \neq n$ and $\mathsf{SA}^{-1}[1]$ otherwise. $\psi(i)$ is computed in time t_ψ, which is at most $O(t_{\mathsf{SA}})$, but usually less. Compressed suffix tree representations also include array $\mathsf{LCP}[1, n]$, which stores the length of the longest common prefix (*lcp*) between consecutive suffixes in lexicographic order, $\mathsf{LCP}[i] = |lcp(T_{\mathsf{SA}[i-1],n}, T_{\mathsf{SA}[i],n})|$ for $i > 1$ and $LCP[1] = 0$. The access time for LCP is t_{LCP}.

We make heavy use of the following complementary operations on bit arrays: $rank(B, i)$ is the number of bits set in $B[1, i]$, and $select(B, j)$ is the position of the j-th 1 in B. Bit vector $B[1, n]$ can be preprocessed to answer both queries in constant time using $o(n)$ extra bits of space [19]. If B contains only m bits set, then the representation of Raman et al. [22] compresses B to $m \log \frac{n}{m} + O(\frac{n \log\log n}{\log n})$ bits of space and retains constant-time *rank* and *select* queries.

3 Compressing LCP Information

Sadakane [25] describes an encoding of the LCP array that uses $2n+o(n)$ bits. The encoding is based on the fact that values $i + \mathsf{LCP}[i]$ are nondecreasing when listed in text position order: Sequence $S = s_1, \ldots, s_{n-1}$, where $s_j = j + \mathsf{LCP}[\mathsf{SA}^{-1}[j]]$, is nondecreasing.

To represent S, Sadakane encodes each $\mathtt{diff}(j) = s_j - s_{j-1}$ in unary: $1\,0^{\mathtt{diff}(j)}$, where $s_0 = 0$ and 0^d denotes repetition of 0-bit d times. This encoding, call it U (similar to *Hgt* [25]), takes at most $2n$ bits. Thus $\mathsf{LCP}[i] = select(U, j + 1) - j - 1$, where $j = \mathsf{SA}[i]$, is computed in time $O(t_{\mathsf{SA}})$.

Let us now consider how to represent U in a yet more space-efficient form, i.e., in $2nH_k \log \frac{1}{H_k} + o(n)$ bits, for small enough k. The result follows from the observation (to be shown below) that the number of 1-bit runs in U is bounded by the number of *runs in* ψ. We call a run in ψ a maximal sequence of consecutive i values where $\psi(i) - \psi(i-1) = 1$ and $T_{\mathsf{SA}[i-1]} = T_{\mathsf{SA}[i]}$, including one preceding i where this does not hold [17]. Note that an area in ψ where the differences are not 1 corresponds to several length-1 runs. Let us call $R \leq n$ the overall number of runs.

We will represent U in run-length encoded form, coding each maximal run of both 0 and 1 bits. We show soon that there are at most R 1-runs, and hence at most R 0-runs (as U starts with a 1). If we encode the 1-run lengths $o_1,\ o_2, \ldots$ and the 0-run lengths $z_1,\ z_2, \ldots$ separately, it is easy to compute $select(U, j)$ by finding the largest r such that $\sum_{i=1}^{r} o_i < j$ and then answering

$select(U, j) = j + \sum_{i=1}^{r} z_i$. This so-called *searchable partial sums problem* is easy to solve. Store bitmap $O[1, n]$ setting the bits at positions $\sum_{i=1}^{r} o_i$, hence $\max\{r,\ \sum_{i=1}^{r} o_i < j\} = rank(O, j-1)$. Likewise, bitmap $Z[1, n]$ representing the z_i's solves $\sum_{i=1}^{r} z_i = select(Z, r)$. Since both O and Z have at most R 1's, O plus Z can be represented using $2R \log \frac{n}{R} + O(\frac{n \log\log n}{\log n})$ bits [22].

We now show the connection between runs in U and runs in ψ. Let us call position i a *stopper* if $i = 1$ or $\psi(i) - \psi(i-1) \neq 1$ or $T_{\mathsf{SA}[i-1]} \neq T_{\mathsf{SA}[i]}$. Hence ψ has exactly R stoppers by the definition of runs in ψ. Say now that a *chain* in ψ is a maximal sequence i, $\psi(i)$, $\psi(\psi(i))$, ... such that each $\psi^j(i)$ is not a stopper except the last one. As ψ is a permutation with just one cycle, it follows that in the path of $\psi^j[\mathsf{SA}^{-1}[1]]$, $0 \leq j < n$, we will find the R stoppers, and hence there are also R chains in ψ [9].

We now show that each chain in ψ induces a run of 1's of the same length in U. Let i, $\psi(i)$, ..., $\psi^\ell(i)$ be a chain. Hence $\psi^j(i) - \psi^j(i-1) = 1$ for $0 \leq j < \ell$. Let $x = \mathsf{SA}[i-1]$ and $y = \mathsf{SA}[i]$. Then $\mathsf{SA}[\psi^j(i-1)] = x+j$ and $\mathsf{SA}[\psi^j(i)] = y+j$. Then $\mathsf{LCP}[i] = |lcp(T_{\mathsf{SA}[i-1],n}, T_{\mathsf{SA}[i],n})| = |lcp(T_{x,n}, T_{y,n})|$. Note that $T_{x+\mathsf{LCP}[i]} \neq T_{y+\mathsf{LCP}[i]}$, and hence $\mathsf{SA}^{-1}[y+\mathsf{LCP}[i]] = \psi^{\mathsf{LCP}[i]}(i)$ is a stopper, thus $\ell \leq \mathsf{LCP}[i]$. Moreover, $\mathsf{LCP}[\psi^j(i)] = |lcp(T_{x+j,n}, T_{y+j,n})| = \mathsf{LCP}[i] - j \geq 0$ for $0 \leq j < \ell$. Now consider $s_{y+j} = y+j+\mathsf{LCP}[\mathsf{SA}^{-1}[y+j]] = y+j+\mathsf{LCP}[\psi^j(i)] = y+j+\mathsf{LCP}[i]-j = y+\mathsf{LCP}[i]$, all equal for $0 \leq j < \ell$. This produces $\ell - 1$ `diff` values equal to 0, that is, a run of ℓ 1-bits in U. By traversing all the chains in the cycle of ψ we sweep S left to right, producing at most R runs of 1's and hence at most R runs of 0's. (Note that even an isolated 1 is a run with $\ell = 1$.) Since $R \leq nH_k + \sigma^k$ for any k [21], we obtain the bound $2nH_k \log \frac{1}{H_k} + O(\frac{n \log\log n}{\log n})$ for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. We emphasize that as $2R \log \frac{n}{R}$ is $2n$ in the worst case, our representation is asymptotically never larger than the original *Hgt*.

4 Next-Smaller and Prev-Smaller Queries

In this section we consider queries *next smaller value* (*NSV*) and *previous smaller value* (*PSV*), and show that they can be solved in sublogarithmic time using only a sublinear number of extra bits on top of the raw data. We make heavy use of these queries in the design of our new compressed suffix tree, and also believe that they can be of independent interest.

Definition 2. *Let $S[1, n]$ be a sequence of elements drawn from a set with a total order $\preceq$ (where one can also define $a \prec b \Leftrightarrow a \preceq b \wedge b \not\preceq a$). We define the query* next smaller value *and* previous smaller value *as follows: $NSV(S, i) = \min\{j,\ (i < j \leq n \wedge S[j] \prec S[i]) \vee j = n+1\}$ and $PSV(S, i) = \max\{j,\ (1 \leq j < i \wedge S[j] \prec S[i]) \vee j = 0\}$, respectively.*

The key idea to solve these queries reminds that for *findopen* and *findclose* operations in balanced parentheses, in particular the recursive version [8]. However, there are several differences because we have to deal with a sequence of generic values, not parentheses.

We will describe the solution for *NSV*, as that for *PSV* is symmetric. For shortness we will write $NSV(i)$ for $NSV(S, i)$. We split $S[1, n]$ into consecutive *blocks* of b values. A position i will be called *near* if $NSV(i)$ is within the same block of i. The first step when solving a *NSV* query will be to scan the values $S[i+1 \ldots b \cdot \lceil i/b \rceil]$, that is from $i+1$ to the end of the block, looking for an $S[j] \prec S[i]$. This takes $O(b)$ time and solves the query for near positions.

Positions that are not near are called *far*. We note that the far positions within a block, $i_1 < i_2 \ldots < i_s$ form a nondecreasing sequence of values $S[i_1] \preceq S[i_2] \ldots \preceq S[i_s]$. Moreover, their *NSV* values form a nonincreasing sequence $NSV(i_1) \geq NSV(i_2) \ldots \geq NSV(i_s)$.

A far position i will be called a *pioneer* if $NSV(i)$ is *not* in the same block of $NSV(j)$, being j the largest far position preceding i (the first far position is also a pioneer). It follows that, if j is the last pioneer preceding i, then $NSV(i)$ is in the same block of $NSV(j) \geq NSV(i)$. Hence, to solve $NSV(i)$, we find j and then scan (left to right) the block $S[\lceil NSV(j)/b \rceil - b + 1 \ldots NSV(j)]$, in time $O(b)$, for the first value $S[j'] \prec S[i]$.

So the problem boils down to efficiently finding the pioneer preceding each position i, and to storing the answers for pioneers. We mark pioneers in a bitmap $P[1,n]$. We note that, since there are $O(n/b)$ pioneers overall [13], P can be represented using $O(\frac{n \log b}{b}) + O(\frac{n \log\log n}{\log n})$ bits of space [22]. With this representation, we can easily find the last pioneer preceding a far position i, as $j = select(P, rank(P,i))$. We could now store the NSV answers for the pioneers in an answer array $A[1,n']$ ($n' = O(n/b)$), so that if j is a pioneer then $NSV(j) = A[rank(P,j)]$. This already gives us a solution requiring $O(\frac{n \log b}{b}) + O(\frac{n \log\log n}{\log n}) + O(\frac{n \log n}{b})$ bits of space and $O(b)$ time. For example, we can have $O(\frac{n}{\log\log n})$ bits of space and $O(\log n \log\log n)$ time.

However, we can do better by recursing on the idea. Instead of storing the answers explicitly in array A, we will form a (virtual) *reduced* sequence $S'[1,2n']$ containing all the pioneer values i and their answers $NSV(i)$. Sequence S' is not explicitly stored. Rather, we set up a bitmap $R[1,n]$ where the selected values in S are marked. Hence we can retrieve any value $S'[i] = S[select(R,i)]$. Again, this can be computed in constant time using $O(\frac{n \log b}{b} + \frac{n \log\log n}{\log n})$ bits to represent R [22].

Because S' is a subsequence of S, it holds that the answers to NSV in S' are the same answers mapped from S. That is, if i is a pioneer in S, mapped to $i' = rank(R,i)$ in S', and $NSV(i)$ is mapped to $j' = rank(R, NSV(i))$, then $j' = NSV(S', i')$, because any value in $S'[i'+1 \ldots j'-1]$ correspond to values within $S[i+1 \ldots NSV(i)-1]$, which by definition of NSV are not smaller than $S[i]$. Hence, we can find $NSV(i)$ for pioneers i by the corresponding recursive query on S', $NSV(i) = select(R, NSV(S', rank(R,i)))$. We are left with the problem of solving queries $NSV(S', i)$.

We proceed again by splitting S' into blocks of b values. Near positions in S' are solved in $O(b)$ time by scanning the block. Recall that S' is not explicitly stored, but rather we have to use *select* on R to get its values from S. For far positions we define again pioneers, and solve NSV on far positions in time $O(b)$ using the answer for the preceding pioneer. Queries for pioneers are solved in a third level by forming the virtual sequence $S''[1,2n'']$, $n'' = O(n'/b) = O(n/b^2)$.

We continue the process recursively for r levels before storing the explicit answers in array $A[1, n^{(r)}]$, $n^{(r)} = O(n/b^r)$. We remark that the P^ℓ and R^ℓ bitmaps at each level ℓ map positions directly to S, not to the reduced sequence of the previous level. This permits accessing the $S^\ell[i]$ values at any level ℓ in constant time, $S^\ell[i] = S[select(R^\ell, i)]$. The pioneer preceding i in S^ℓ is found by first mapping to S with $i' = select(R^\ell, i)$, then finding the preceding pioneer directly in the domain of S, $j' = select(P^\ell, rank(P^\ell, i'))$, and finally mapping the pioneer back to S^ℓ by $j = rank(R^\ell, j')$.

Let us now analyze the time and space of this solution. Because we pay $O(b)$ time at each level and might have to resort to the next level in case our position is far, the total time is $O(rb)$ because the last level is solved in constant time. As for the space, all we store are the P^ℓ and R^ℓ bitmaps, and the final array A. Array A takes $O(\frac{n \log n}{b^r})$ bits. As there are $O(n/b^\ell)$ elements in S^ℓ, both P^ℓ and R^ℓ require $O(\frac{n}{b^\ell}\log(b^\ell) + \frac{n \log\log n}{\log n})$ bits of space (actually P^ℓ is about half the size of R^ℓ). The sum of all the P^ℓ and R^ℓ takes order of

$$\sum_{1 \leq \ell \leq r} \left(\frac{n}{b^\ell} \log(b^\ell) + \frac{n \log\log n}{\log n} \right) \;\leq\; n \log b \left(\sum_{\ell \geq 1} \frac{\ell}{b^\ell} \right) + r\, \frac{n \log\log n}{\log n} \;=\; O\left(\frac{n \log b}{b} + r\, \frac{\log\log n}{\log n} \right).$$

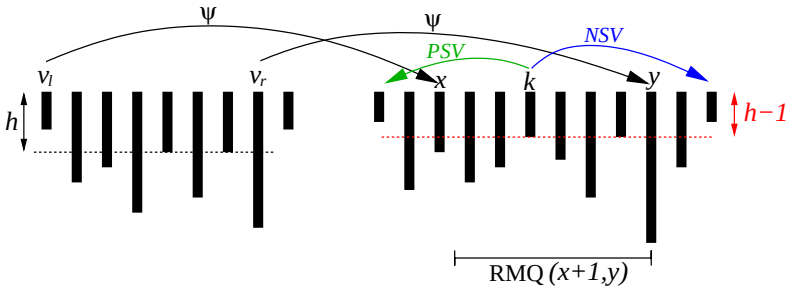


Fig. 1. Left: Illustration to the representation of suffix tree nodes. The lengths of the bars indicate the LCP values. All leaves in the subtree rooted at $v = [v_l, v_r]$ share a longest common prefix of length at least h. Right: Schematic view of the SLink operation. From v, first follow ψ, then perform an RMQ to find an $(h-1)$-index k, and finally locate the defining points of the desired interval by a PSV/NSV query from k.

We now state the main result of this section (rest of proof in Appendix A).

Theorem 1. *Let $S[1,n]$ be a sequence of elements drawn from a set with a total order, such that access to any $S[i]$ and any comparison $S[i] \prec S[j]$ can be computed in constant time. Then, for any $1 \le r, b \le n$, it is possible to build a data structure on S taking $O(\frac{n \log b}{b} + r\frac{n \log\log n}{\log n} + \frac{n \log n}{b^r})$ bits, so that queries NSV and PSV can be solved in worst-case time time $O(rb)$. In particular, for any $f(n) = O(\frac{\log n}{\log\log n})$, one can achieve $O(\frac{n}{f(n)})$ bits of extra space and $O(f(n) \log\log n)$ time.*

Note that, if one is willing to spend $4n + o(n)$ bits of extra space, the operations can be solved in constant time. The idea is to reduce *PSV* and *NSV* queries to $O(1)$ *findopen* and *findclose* operations in balanced parentheses [8]. For *NSV*, for $1 \le i \le n+1$ in this order, write a '(' and then x ')'s if there are x cells $S[j]$ for which $NSV(j) = i$. The resulting sequence B is balanced if a final ')' is appended, and $NSV(i)$ can be obtained by $rank(B, findclose(B, select(B, i)))$, where a 1 in B represents '('. The solution is symmetric for *PSV*, needing other $2n + o(n)$ bits.

5 An Entropy-Bounded Compressed Suffix Tree

Let v be a node in the (virtual) suffix tree $\mathcal{S}$ for text $T_{1,n}$. As in previous works [1, 4, 23], we represent v by an interval $[v_l, v_r]$ in SA such that $\mathsf{SA}[v_l, v_r]$ are exactly the leaves in $\mathcal{S}$ that are in the subtree rooted at v. Let us first consider internal nodes, so $v_l < v_r$. Because $\mathcal{S}$ does not contain unary nodes, it follows from the definition of LCP that at least one entry in $\mathsf{LCP}[v_l + 1, v_r]$ is equal to the string-depth h of v; such a position is called *h-index* of $[v_l, v_r]$. We further have $\mathsf{LCP}[v_l] < h$, $\mathsf{LCP}[i] \ge h$ for all $v_l < i \le v_r$, and $\mathsf{LCP}[v_r + 1] < h$. Fig. 1 (left) illustrates. We state the easy yet fundamental

Lemma 1. *Let $[v_l, v_r]$ be an interval in SA that corresponds to an internal node v in $\mathcal{S}$. Then the string-depth of v can be obtained as $h = \mathsf{LCP}(k)$, where $k = \textsc{rmq}_{\mathsf{LCP}}(v_l + 1, v_r)$.*

For leaves $v = [v_l, v_l]$, the string-depth of v is simply given by $n - \mathsf{SA}[v_l] + 1$.

5.1 Range Minimum Queries in Sublinear Space

As Lemma 1 suggests, we wish to preprocess LCP such that $\textsc{rmq}_{\mathsf{LCP}}$ can be answered in sublogarithmic time, using $o(n)$ bits of additional space. A well-known strategy [6, 25] divides LCP iteratively

into *blocks* of decreasing size $n > b_1 > b_2 > \cdots > b_r$. On level i, $1 \le i \le r$, compute all answers to $\text{RMQ}_{\mathsf{LCP}}$ that exactly span over blocks of size b_i, but not over blocks of size b_{i-1} (set $b_0 = n$ for handling the border case). This takes $O(\frac{n}{b_i}\log(\frac{b_{i-1}}{b_i})\log(b_{i-1}))$ bits of space if the answers are stored relative to the beginning of the blocks on level $i-1$, and if we only precompute queries that span 2^j blocks for some $j \le \lfloor\log(\frac{b_{i-1}}{b_i})\rfloor$ (this is sufficient because each query can be decomposed into at most 2 possibly overlapping sub-queries whose lengths are a power of 2).

A general range minimum query is then decomposed into at most $2r+1$ non-overlapping sub-queries $q_1, \ldots, q_{2r+1}$ such that q_1 and q_{2r+1} lie completely inside of blocks of size b_r, q_2 and q_{2r} exactly span over blocks of size b_r, and so on. q_1 and q_{2r+1} are solved by scanning in time $O(b_r)$,[2] and all other queries can be answered by table-lookups in total time $O(r)$. The final answer is obtained by comparing at most $2r+1$ minima.

The next lemma gives a general result for RMQs using $o(n)$ extra space (proof in Appendix A).

Lemma 2. *Having constant-time access to elements in an array $A[1,n]$, it is possible to answer range minimum queries on A in time $O(f(n)(\log f(n))^2)$ using $O(\frac{n}{f(n)})$ bits of space, for any $f(n) = \Omega(\log^{[r]} n)$ and any constant r, where $\log^{[r]} n$ denotes r applications of $\log$ to n.*

5.2 Suffix-Tree Operations

Now we have all the ingredients for navigating in the suffix tree. The operations are described in the following; the intuitive reason why an RMQ is often followed by a PSV/NSV-query is that the RMQ gives us an h-index of the (yet unknown) interval, and the PSV/NSV takes us to the delimiting points of this interval. Apart from t_{SA}, t_{LCP}, and t_ψ, we denote by t_{RMQ} and t_{PNSV} the time to solve, respectively, RMQs or NSV/PSV queries (both on LCP from now on, hence they will be multiplied by t_{LCP}).

ROOT/COUNT/ANCESTOR: ROOT() returns the interval $[1,n]$, COUNT(v) is simply $v_r - v_l + 1$, ANCESTOR(v,w) is true iff $w_l \le v_l \le v_r \le w_r$. All these can be computed in $O(1)$ time.

SDEPTH(v)/LOCATE(v): According to Lemma 1, SDEPTH(v) can be computed in time $O(t_{\text{RMQ}} \cdot t_{\mathsf{LCP}})$ for internal nodes, and both operations need time $O(t_{\mathsf{SA}})$ for leaves. One knows in constant time that $v = [v_l, v_r]$ is a leaf iff $v_l = v_r$.

PARENT(v): If v is the root, return NULL. Otherwise, since the suffix tree is compact, we must have that the string-depth of PARENT(v) is either $\mathsf{LCP}[v_l]$ or $\mathsf{LCP}[v_r+1]$, whichever is greater [23]. So, by setting $k =$ **if** $\mathsf{LCP}[v_l] > \mathsf{LCP}[v_r+1]$ **then** v_l **else** v_r+1, the parent interval of v is $[PSV(k), NSV(k)-1]$. Time is $O(t_{\text{PNSV}} \cdot t_{\mathsf{LCP}})$.

FCHILD(v): If v is a leaf, return NULL. Otherwise, because the minima in $[v_l, v_r]$ are v's h-indices [6], the first child of v is given by $[v_l, \text{RMQ}(v_l+1, v_r)-1]$, assuming that RMQs always return the leftmost minimum in the case of ties (which is easy to arrange). Time is $O(t_{\text{RMQ}} \cdot t_{\mathsf{LCP}})$.

NSIBLING(v): First move to the parent of v by $w =$ PARENT(v). If $v_r = w_r$, return NULL, since v does not have a next sibling. If $v_r + 1 = w_r$, v's next sibling is a leaf, so return $[w_r, w_r]$. Otherwise, return $[v_r+1, \text{RMQ}(v_r+2, w_r)-1]$. The overall time is $O((t_{\text{RMQ}} + t_{\text{PNSV}}) \cdot t_{\mathsf{LCP}})$.

SLINK(v): If v is the root, return NULL. Otherwise, first follow the suffix links of the leaves v_l and v_r, $x = \psi(v_l)$ and $y = \psi(v_r)$. Then locate an h-index of the target interval by $k = \text{RMQ}(x+1, y)$;

[2] The constant-time solutions [25, 6] also solve q_1 and q_{2r+1} by accessing tables that require $\Theta(n)$ bits.

see Lemma 7.5 in [1] (the first character of all strings in $\{T_{\mathsf{SA}[i],n} : v_l \le i \le v_r\}$ is the same, so the h-indices in $[v_l, v_r]$ appear also as $(h-1)$-indices in $[\psi(v_l), \psi(v_r)]$). The final result is then given by $[PSV(k), NSV(k) - 1]$. Time is $O(t_\psi + (t_{\text{PNSV}} + t_{\text{RMQ}}) \cdot t_{\mathsf{LCP}}))$. Fig. 1 (right) illustrates.

SLink$^i(v)$: Same as above with $x = \psi^i(v_l)$ and $y = \psi^i(v_r)$. If the first Letter of x and y are different, then the answer is Root. Otherwise we go on with k as before. Computing ψ^i can be done in $O(t_{\mathsf{SA}})$ time using $\psi^i(v) = \mathsf{SA}^{-1}[\mathsf{SA}[v] + i]$ [23]. Time is thus $O(t_{\mathsf{SA}} + (t_{\text{PNSV}} + t_{\text{RMQ}}) \cdot t_{\mathsf{LCP}})$.

LCA(v, w): If one of v or w is an ancestor of the other, return this ancestor node. Otherwise, w.l.o.g., assume $v_r < w_l$. The h-index of the target interval is given by an RMQ between v and w [25]: $k = \text{RMQ}(v_r + 1, w_l)$. The final answer is again $[PSV(k), NSV(k) - 1]$. Time is $O((t_{\text{RMQ}} + t_{\text{PNSV}}) \cdot t_{\mathsf{LCP}})$.

Child(v, a): If v is a leaf, return null. Otherwise, the minima in $\mathsf{LCP}[v_l + 1, v_r]$ define v's child-intervals, so we need to find the position $p \in [v_l + 1, v_r]$ where $\mathsf{LCP}[p] = \min_{i \in [v_l+1, v_r]} \mathsf{LCP}[i]$, and $T_{\mathsf{SA}[p]+\mathsf{LCP}[p]} = \text{Letter}([p, p], \mathsf{LCP}[p]) = a$. Then the final result is given by $[p, \text{RMQ}(p + 1, v_r) - 1]$, or null if there is no such position p. To find this p, split $[v_l, v_r]$ into three sub-intervals $[v_l, x - 1], [x, y - 1], [y, v_r]$, where x (y) is the first (last) position in $[v_l, v_r]$ where a block of size b_r starts (b_r is the smallest block size for precomputed RMQs, recall Sect. 5.1). Intervals $[v_l, x - 1]$ and $[y, v_r]$ can be scanned for p in time $O(t_{\text{RMQ}} \cdot (t_{\mathsf{LCP}} + t_{\mathsf{SA}}))$. The big interval $[x, y - 1]$ can be binary-searched in time $O(\log \sigma \cdot t_{\mathsf{SA}})$, provided that we also store exact median positions of the minima in the precomputed RMQs [25] (within the same space bounds). The only problem is how these precomputations are carried out in $O(n)$ time, as it is not obvious how to compute the exact median of an interval from the medians in its left and right half, respectively. However, a solution to this problem exists [7, Sect. 3.2]. Overall time is $O((t_{\mathsf{LCP}} + t_{\mathsf{SA}}) \cdot t_{\text{RMQ}} + \log \sigma \cdot t_{\mathsf{SA}})$.

Letter(v, i): If $i = 1$ we can easily solve the query in constant time with very little extra space. Mark in a bitmap $C[1, n]$ the first suffix in SA starting with each different letter, and store in a string $L[1, \sigma]$ the different letters that appear in $T_{1,n}$ in alphabetical order. Hence, if $v = [v_l, v_r]$, $\text{Letter}(v, 1) = L[rank(C, v_l)]$. L requires $O(\sigma \log \sigma)$ bits and C, represented as a compressed bitmap [22], requires $O(\sigma \log \frac{n}{\sigma} + \frac{n \log \log n}{\log n})$ bits of space. Hence both add up to $O(\sigma \log n + \frac{n \log \log n}{\log n})$ bits. Now, for $i > 1$, we just use $\text{Letter}(v, i) = \text{Letter}(\psi^{i-1}(v_l), 1)$, in time $O(\min(t_{\mathsf{SA}}, i \cdot t_\psi))$. We remark that structures L and C are already present, in one form or another, in all compressed text indexes implementing SA [10, 24, 5].

TDepth(v): Tree-depth can usually be maintained while performing the other operations (apart from LCA and LAQs): it increases by 1 in FChild and Child, decreases by 1 (or i) in Parent and SLink (SLink^i), becomes d in LAQt$(\cdot, d)$, and stays the same otherwise.

However, there is also a direct way to support TDepth, using other $2nH_k \log \frac{1}{H_k}$ bits of space. The idea is similar to Sadakane's representation of LCP [25]: the key insight is that the tree depth can decrease by at most 1 if we move from suffix $T_{i,n}$ to $T_{i+1,n}$ (i.e., when following ψ). Define $\mathsf{TDE}[1, n]$ such that $\mathsf{TDE}[i]$ holds the tree-depth of the LCA of leaves $\mathsf{SA}[i]$ and $\mathsf{SA}[i - 1]$ (similar to the definition of LCP). Then the sequence $(\mathsf{TDE}[\psi^k(\mathsf{SA}^{-1}[1])] + k)_{k=0,1,\ldots,n-1}$ is nondecreasing and in the range $[1, n]$, and can hence be stored using $2n + o(n)$ bits. Further, the repetitions appear in the same way as in *Hgt* (Sect. 3), so the resulting sequence can be compressed to $2nH_k \log \frac{1}{H_k}$ bits using the same mechanism as for LCP. The time is thus $O(t_{\mathsf{LCP}})$.

LAQs(v, d): Let $u = [u_l, u_r] = \text{LAQs}(v, d)$ denote the (yet unknown) result. Because u is an ancestor of v, we must have $u_l \le v_l$ and $v_r \le u_r$. We further know that $\mathsf{LCP}[i] \ge d$ for all $u_l < i \le u_r$. Thus, u_l is the largest position in $[1, v_l]$ with $\mathsf{LCP}[u_l] < d$. So the search for u_l can

be conducted in a binary manner by means of RMQs: Letting $k = \text{RMQ}(\lfloor v_l/2 \rfloor, v_l)$, we check if $\mathsf{LCP}[k] \geq d$. If so, u_l cannot be in $[\lfloor v_l/2 \rfloor, v_l]$, so we continue searching in $[1, \lfloor v_l/2 \rfloor - 1]$. If not, we know that u_l must be in $[\lfloor v_l/2 \rfloor, v_l]$, so we continue searching in there. The search for u_r is handled symmetrically. Total time is $O(\log n \cdot t_{\text{RMQ}} \cdot t_{\mathsf{LCP}})$.

$\text{LAQ}_\text{T}(v, d)$: The same idea as for LAQs can be applied here, using the array TDE instead of LCP, and RMQs on TDE. Time is also $O(\log n \cdot t_{\text{RMQ}} \cdot t_{\mathsf{LCP}})$.

6 Discussion

The final performance of our compressed suffix tree (CST) depends on the compressed full-text index used to implement SA. Among the best choices we have Sadakane's compressed suffix array (SCSA) [24], which is not so attractive for its $O(n \log\log \sigma)$ extra bits of space in a context where we are focusing on using $o(n)$ extra space. The alphabet-friendly FM-index (AFFM) [5] gives the best space, but our CST over AFFM is worse than Russo et al.'s CST (RCST) [23] both in time and space. Instead, we focus on using Grossi et al.'s compressed suffix array (GCSA) [10], which is larger than AFFM but lets our CST achieve better times than RCST. (Interestingly enough, RCST does not benefit from using the larger GCSA.) Our resulting CST is a space/time tradeoff between Sadakane's CST (SCST) [25] and RCST. Within this context, it makes sense to consider SCST on top of GCSA, to remove the huge $O(n \log\log \sigma)$ extra space of SCSA.

GCSA uses $|GCSA| = (1+\frac{1}{\epsilon})nH_k + O(\frac{n \log\log n}{\log_\sigma n})$ bits of space for any $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, and offers times $t_\psi = O(1)$ and $t_{\mathsf{SA}} = O(\log^\epsilon n \log^{1-\epsilon} \sigma)$. On top of $|GCSA|$, SCST needs $6n + o(n)$ bits, whereas our CST needs $2nH_k \log \frac{1}{H_k} + o(n)$ extra bits. Our CST times are $t_{\mathsf{LCP}} = t_{\mathsf{SA}}$, whereas t_{RMQ} and t_{PNSV} depend on how large is $o(n)$. Instead, RCST needs $|AFFM| + o(n)$ bits, where $|AFFM| = nH_k + O(\frac{n \log\log n}{\log_\sigma n}) + O(\frac{n \log n}{\gamma})$ bits, for some $\gamma = \omega(\log_\sigma n)$, to maintain the extra space $o(n \log \sigma)$. AFFM offers times $t_\psi = O(1 + \frac{\log \sigma}{\log\log n})$ and $t_{\mathsf{SA}} = O(\gamma(1 + \frac{\log \sigma}{\log\log n}))$. In addition, RCST uses $o(n) = O(\frac{n \log n}{\delta})$ bits for a parameter $\delta = \omega(\log_\sigma n)$.

An exhaustive comparison is complicated, as it depends on ϵ, γ, δ, the nature of the $o(n)$ extra bits in our CST, σ, etc. In general, our CST loses to RCST if they use the same amount of space, yet our CST can achieve sublogarithmic times by using some extra space, whereas RCST cannot. We opt for focusing on a particular setting that exhibits this space/time tradeoff. The reader can easily derive other settings. We focus on the case $\sigma = O(1)$ and all extra spaces not related to entropy limited to $O(\frac{n}{\log^{\epsilon'} n})$ bits, for constant $0 < \epsilon' < 1$ (so $f(n) = \log^{\epsilon'} n$ in Thm. 1 and Lemma 2). Thus, our times are $t_{\text{RMQ}} = \log^{\epsilon'} n (\log\log n)^2$ and $t_{\text{PNSV}} = \log^{\epsilon'} n \log\log n$. RCST's γ and δ are $O(\log^{1+\epsilon'} n)$. Table 1 (Appendix B) shows a comparison under this setting. The first column also summarizes the general complexities of our operations, with no assumptions on σ nor extra space except $t_\psi \leq t_{\mathsf{SA}} = t_{\mathsf{LCP}}$, as these are intrinsic of our structure.

Clearly SCST is generally faster than the others, but it requires $6n + o(n)$ non-compressible extra bits on top of $|CSA|$. RCST is smaller than the others, but its time is typically $O(\log^{1+\epsilon'} n)$ for some constant $0 < \epsilon' < 1$. The space of our CST is in between, with typical time $O(\log^\lambda n)$ for any constant $\lambda > \epsilon + \epsilon'$. This can be sublogarithmic when $\epsilon + \epsilon' < 1$. To achieve this, the permissible space in the entropy-related part is larger than $2(1 + \log \frac{1}{H_k})nH_k$. With less than that space our CST is slower than the smaller RCST, but using more than that space our CST achieves sublogarithmic times (except for level ancestor queries), being the only compressed suffix tree achieving it within $o(n)$ extra space.

Acknowledgments. JF wishes to thank Volker Heun and Enno Ohlebusch for interesting discussions on this subject. GN was partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

References

1. M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.
2. A. Apostolico. *The myriad virtues of subword trees*, pages 85–96. Combinatorial Algorithms on Words. NATO ISI Series. Springer-Verlag, 1985.
3. O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 14(3):344–370, 1993.
4. R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Proc. 33rd ICALP*, LNCS 4051, pages 358–369, 2006.
5. 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.
6. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, LNCS 4614, pages 459–470, 2007.
7. J. Fischer and V. Heun. Range median of minima queries, super cartesian trees, and text indexing. Manuscript. Available at `www.bio.ifi.lmu.de/~fischer/fischer10range.pdf`, 2007.
8. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th CPM*, LNCS 3109, pages 159–172, 2004.
9. R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th CPM*, LNCS 4580, pages 216–227, 2007.
10. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
11. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. on Computing*, 35(2):378–407, 2006.
12. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
13. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
14. J. Kärkkäinen and S. Rao. *Algorithms for Memory Hierarchies*, chapter 7: Full-text indexes in external memory, pages 149–170. LNCS 2625. Springer, 2003.
15. P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In *Proc. 14th SPIRE*, LNCS 4726, pages 184–194, 2007.
16. S. Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.
17. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.
18. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. of the ACM*, 48(3):407–430, 2001.
19. I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
20. I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *J. of Algorithms*, 39(2):205–222, 2001.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
22. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
23. L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th LATIN*, LNCS, 2008. To appear.
24. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.
25. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2007. To appear. DOI 10.1007/s00224-006-1198-x.

A Proofs

Theorem 1. The general formula for any r, b has been obtained thruoghout the section. As for the formulas in terms of $f(n)$, let us set the space limit to $O(\frac{n}{f(n)})$. Then $\frac{n \log b}{b} = O(\frac{n}{f(n)})$ implies $b = \Omega(f(n) \log f(n))$. Also, $\frac{n \log n}{b^r} = O(\frac{n}{f(n)})$ implies $r \geq \frac{\log \log n + \log f(n) - O(1)}{\log b}$. Hence $rb \geq \frac{b}{\log b} (\log \log n + \log f(n) - O(1))$. Thus it is best to minimize b. By setting $b = f(n) \log f(n)$, we get $rb = \frac{f(n) \log f(n)}{\log f(n) + \log \log f(n)} (\log \log n + \log f(n) - O(1)) = \Theta(f(n)(\log \log n + \log f(n)))$. The final constraint is $r \frac{n \log \log n}{\log n} = O(\frac{n}{f(n)})$, which, by substituting $r = \frac{\log \log n + \log f(n)}{\log b}$ and considering that $b = \Omega(f(n) \log f(n))$, yields the condition $f(n) = O(\frac{\log n}{\log \log n})$. Thus $\log \log n + \log f(n) = O(\log \log n)$. □

Lemma 2. We use $r + 1 = O(1)$ levels $1 \ldots r + 1$, so it is sufficient that $\frac{n}{b_i} \log^2 b_{i-1} = O(\frac{n}{f(n)})$ for all $1 \leq i \leq r + 1$, where $b_0 = n$. From the condition $\frac{n}{b_1} \log^2 b_0 = O(\frac{n}{f(n)})$ we get $b_1 = \Theta(f(n) \log^2 n)$ (the smallest possible b_i values are best). From $\frac{n}{b_2} \log^2 b_1 = O(\frac{n}{f(n)})$ we get $b_2 = \Theta(f(n) \log^2 b_1) = \Theta(f(n)(\log f(n) + \log \log n)^2)$. In turn, from $\frac{n}{b_3} \log^2 b_2 = O(\frac{n}{f(n)})$ we get $b_3 = \Theta(f(n) \log^2 b_2) = \Theta(f(n)(\log f(n) + \log \log \log n)^2)$. This continues until $b_{r+1} = \Theta(f(n) \log^2 b_r) = \Theta(f(n)(\log f(n) + \log^{[r+1]} n))^2 = \Theta(f(n) \log^2 f(n))$. □

B Tables

Operation	Our suffix tree		Other suffix trees	
	General	over GCSA [10]	SCST [25]	RCST [23]
ROOT,COUNT,ANCESTOR	1	1	1	1
LOCATE	t_{SA}	$\log^{\epsilon} n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
SDEPTH	$t_{\mathsf{SA}} \cdot t_{\text{RMQ}}$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
PARENT	$t_{\mathsf{SA}} \cdot t_{\text{PNSV}}$	$\log^{\epsilon+\epsilon'} n \log \log n$	1	$\log^{1+\epsilon'} n$
FCHILD	$t_{\mathsf{SA}} \cdot t_{\text{RMQ}}$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	1	$\log^{1+\epsilon'} n$
NSIBLING	$t_{\mathsf{SA}}(t_{\text{RMQ}} + t_{\text{PNSV}})$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	1	$\log^{1+\epsilon'} n$
SLINK,LCA	$t_{\mathsf{SA}}(t_{\text{RMQ}} + t_{\text{PNSV}})$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	1	$\log^{1+\epsilon'} n$
SLINK^i	$t_{\mathsf{SA}}(t_{\text{RMQ}} + t_{\text{PNSV}})$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
CHILD	$t_{\mathsf{SA}}(t_{\text{RMQ}} + \log \sigma)$	$\log^{\epsilon+\epsilon'} n (\log \log n)^2$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n \log \log n$
LETTER	t_{SA}	$\log^{\epsilon} n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
TDEPTH	t_{SA} (*)	$\log^{\epsilon} n$	1	$\log^{2+2\epsilon'} n$
LAQS	$t_{\mathsf{SA}} \cdot t_{\text{RMQ}} \cdot \log n$	$\log^{1+\epsilon+\epsilon'} n (\log \log n)^2$	Not supported	$\log^{1+\epsilon'} n$
LAQT	$t_{\mathsf{SA}} \cdot t_{\text{RMQ}} \cdot \log n$ (*)	$\log^{1+\epsilon+\epsilon'} n (\log \log n)^2$	1	$\log^{2+2\epsilon'} n$

(*) Our CST needs other $2nH_k \log \frac{1}{H_k} + o(n)$ extra bits to implement TDEPTH and LAQT.

Table 1. Comparison between ours and alternative compressed suffix trees. The column labeled 'General' assumes $t_\psi \leq t_{\mathsf{SA}} = t_{\mathsf{LCP}}$. All other columns further assume $\sigma = O(1)$, and that the extra spaces is $O(\frac{n}{\log^{\epsilon'} n})$.