Faster Entropy-Bounded Compressed Suffix Trees[☆]

Johannes Fischer[a,1], Veli Mäkinen[b,2], Gonzalo Navarro[c,3]

[a] *Center for Bioinformatics (ZBIT), University of Tübingen.* fischer@informatik.uni-tuebingen.de.
[b] *Department of Computer Science, University of Helsinki, Finland.* vmakinen@cs.helsinki.fi.
[c] *Department of Computer Science, University of Chile.* gnavarro@dcc.uchile.cl.

Abstract

Suffix trees are among the most important data structures in stringology, with a number of applications in flourishing areas like bioinformatics. Their main problem is space usage, which has triggered much research striving for compressed representations that are still functional. A smaller suffix tree representation could fit in a faster memory, outweighing by far the theoretical slowdown brought by the space reduction. We present a novel compressed suffix tree, which is the first achieving at the same time sublogarithmic complexity for the operations, and space usage that asymptotically goes to zero as the entropy of the text does. The main ideas in our development are compressing the longest common prefix information, totally getting rid of the suffix tree topology, and expressing all the suffix tree operations using range minimum queries and a novel primitive called next/previous smaller value in a sequence. Our solutions to those operations are of independent interest.

Key words: Suffix trees, compressed data structures, range minimum queries.

1. Introduction

Suffix trees are probably the most important structure ever invented in stringology. They have been said to have a myriad of virtues [3], and also have a myriad of applications in many areas, most prominently bioinformatics [26]. 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 [34]. 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 [28] and it is an active area of research [33], it is always faster to operate in main memory.

[☆] A conference version of this paper appeared in [20].

[1] Worked on this article while at LMU München and while visiting University of Chile.
[2] Funded by the Academy of Finland under grant 119815.
[3] Partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

This situation has stimulated research on compressed representations of suffix trees, which operate without need of decompression. That is, the goal is not traditional data compression, where data must be decompressed before accessing it, but compressed data structures, which operate within reduced space. 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 [41], 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 [40, 25].

The first fully-functional (i.e., supporting a thorough standard set of operations) compressed suffix tree is due to Sadakane [45]. It builds on top of a compressed suffix array [44] that uses $\frac{1}{\epsilon}nH_0 + O(n \log\log \sigma)$ bits of space, where H_0 is the zero-order empirical 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 [24], 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 [38] and $0 < \alpha < 1$ is any constant. However, the $O(\log^\epsilon n)$ time complexities become $O(\log^{\frac{\epsilon}{1-\epsilon}}_\sigma n \log \sigma)$ [24, Theorem 4.1]. In addition, the extra $6n$ bits in the space complexity remain, despite any reduction we can achieve in the compressed suffix array. The $6n$ bit-term can be split into $2n$ bits to represent (with a bitmap called H) the longest common prefix information (LCP, called *Hgt* in that work [45]), plus $4n$ bits to represent the suffix tree topology with parentheses. Many operations are solved via constant-time rank- and select queries in the parentheses sequence.

Russo et al. [43] 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 [14]. 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, which removes 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 $nH_k(2 \log \frac{1}{H_k} + \frac{1}{\epsilon} + O(1)) + o(n \log \sigma)$ 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 incompressible $6n$ bits in Sadakane's structure. Indeed this is a crude upper bound that makes sense only when $H_k < 1$; see point (1) next for a more refined formula. At worst, our space is still $2n + nH_k(\frac{1}{\epsilon} + O(1)) + o(n \log \sigma)$ bits, that is, we need $2n$ instead of $6n$ bits.

Our solution builds on two novel algorithmic ideas to improve Sadakane's compressed suffix tree. The first takes care (with success depending on the compressibility of the

text, as explained) of the $2n$ bits of the LCP information, whereas the second removes the $4n$ bits of the tree topology at the price of converting constant times into sublogarithmic.

1. We show that array H, which encodes LCP information in $2n$ bits [45, Section 4.1], actually contains $2R$ runs, where R is the number of *runs in* ψ [41]. We show how to run-length compress H into $2R \log \frac{n}{R} + O(R) + o(n)$ 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 [36], although sometimes it is extremely pessimistic (and is meaningful only for $H_k < 1$, as obviously $R \leq n$). This gives the $nH_k(2 \log \frac{1}{H_k} + O(1))$ upper bound to store H, and in any case the actual space is $\leq 2n$ bits.
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 range minimum queries (RMQs) on LCP (represented with H), plus a new type of queries called "Next/Previous Smaller Value" (NSV/PSV). An RMQ from i to $j \geq i$ over a sequence $S[1, n]$ of numbers asks for $\mathrm{RMQ}_S(i, j) := \mathrm{argmin}_{i \leq \ell \leq j} S[\ell]$. In an NSV-/PSV-query, we wish to find the first cell in S following/preceding i whose value is smaller than $S[i]$. We show how to solve these queries in sublogarithmic time while spending only $o(n)$ extra bits of space on top of S.

These latter operations have independent interest. Computing RMQs is a well studied problem, including very recent findings related to achieving constant time with $O(n)$ bits of extra space [6, 2, 4, 16, 45, 46, 17]. We give sublogarithmic-time solutions using $o(n)$ bits of extra space, leaving open the challenge of achieving constant time within this space.

Interest in PSV/NSV-like queries is more recent. It has been considered earlier in parallel computing [5], yet not in the static scenario. This latter case is our focus. We show that PSV/NSV can be solved in constant time using $\Theta(n)$ bits of space, and in sublogarithmic time using $o(n)$ bits of space. The challenge of achieving constant time with sublinear space remains open.

Crochemore et al. [11, 10] have studied a related problem called "Range Next Value" (RNV), where on a given range $[i, j]$ and value ℓ one needs to find the cell in $S[i, j]$ whose value is smallest among those greater than or equal to ℓ. The best constant time solution to RNV requires $O(n^{1+\epsilon})$ space, for any constant $\epsilon > 0$ [10]. It is an interesting open question whether RNV could be solved in space close to that of our PSV/NSV solutions. That would have direct consequences for several problems, for example in solving position-restricted pattern searches [37].

The outline of the paper is as follows. In Section 2 we explain some basic concepts on suffix trees, compressed text indexes, and compact data structures. In Section 3 we show how LCP can be compressed to less than $2n$ bits when T is compressible, while retaining constant-time access. In Section 4 we give sublogarithmic-time solutions to PSV/NSV queries using $o(n)$ extra space, and a constant-time solution using $4n$ bits of space. Section 5 gives novel sublogarithmic time solutions to RMQs using $o(n)$ bits of space. Section 6 shows how all the suffix tree operations can be solved in compressed space using the new primitives. Finally, Section 7 discusses the theoretical achievements in the context of related work on compressed suffix trees, and Section 8 discusses possible practical implementations of our theoretical proposal.

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 [3, 26]. 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 [43]). 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.
- 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): the alphabetically first child of v.
- NSIBLING(v): the alphabetically 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* [41, 44, 24, 14], 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 $\mathsf{LCP}[1] = 0$. 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 [39]. If B contains only m bits set, then the representation of Raman et al. [42] compresses B to $m \log \frac{n}{m} + O(m + \frac{n \log\log n}{\log n})$ bits of space and retains constant-time *rank* and *select* queries.

By $\log x$ we mean $\log_2 x$. By $\log^t h(n)$ we mean $(\log h(n))^t$.

3. Compressing LCP Information

Sadakane [45] 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 H (following Sadakane [45, Section 4.1]), takes at most $2n$ bits. Thus $\mathsf{LCP}[i] = select(H, j+1) - 2j + 1$, where $j = \mathsf{SA}[i]$, is computed in time $O(t_{\mathsf{SA}})$.

Let us now consider how to represent H in a yet more space-efficient form, that is, in $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits, for small enough k. The result follows from the observation (to be shown below, in Lemma 1) that the number of 1-bit runs in H 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 [36]. 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 in ψ.

We will represent H 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 H 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 (cf. [12, Section 3.2]), it is easy to compute $select(H, j)$ by finding the largest r such that $\sum_{i=1}^{r} o_i < j$ and then answering $select(H, j) = j + \sum_{i=1}^{r} z_i$. 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 $2R \log \frac{n}{R} + O(R + \frac{n \log\log n}{\log n})$ bits [42].

We now prove the connection between runs in H and runs in ψ, and conclude by formalizing the result of this section.

Lemma 1. *Let H and R be as defined in this section. Then bitmap H has at most R runs of 1's (where even isolated 1's count as a run).*

Proof. 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. Now say that a *chain* in ψ is a maximal sequence $i, \psi(i), \psi(\psi(i)), \ldots$ 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 [23].

We now show that each chain in ψ induces a run of 1's of the same length in H. Let i, $\psi(i)$, ..., $\psi^\ell(i)$ be a chain. Hence $\psi^j(i) - \psi^j(i-1) = 1$ for $0 \le j < \ell$. Let $x = \mathsf{SA}[i-1]$ and $y = \mathsf{SA}[i]$. Then $\mathsf{SA}[\psi^j(i-1)] = x + j$ and $\mathsf{SA}[\psi^j(i)] = y + j$. Also, $\mathsf{LCP}[i] = |lcp(T_{\mathsf{SA}[i-1],n}, T_{\mathsf{SA}[i],n})| = |lcp(T_{x,n}, T_{y,n})|$. 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 $\ell \le \mathsf{LCP}[i]$. Moreover, $\mathsf{LCP}[\psi^j(i)] = |lcp(T_{x+j,n}, T_{y+j,n})| = \mathsf{LCP}[i] - j \ge 0$ for $0 \le j < \ell$. 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 $0 \le j < \ell$. This produces $\ell - 1$ `diff` values equal to 0, that is, a run of ℓ 1-bits in H. By traversing all the chains in the cycle of ψ we sweep S left to right, producing at most R runs of 1's. □

Theorem 1. *The* LCP *array of a text of length n whose ψ function has R runs, can be represented using $2R\log\frac{n}{R} + \Theta(R) + O(\frac{n\log\log n}{\log n}) = nH_k(2\log\frac{1}{H_k} + \Theta(1)) + O(\frac{n\log\log n}{\log n})$ bits, for any $k \le \alpha\log_\sigma n$ and any constant $0 < \alpha < 1$. Access to $\mathsf{LCP}[i]$ takes constant time given the value of $\mathsf{SA}[i]$, the corresponding suffix array cell.*

Proof. We showed how LCP is represented using H (so that $\mathsf{SA}[i]$ is needed to retrieve $\mathsf{LCP}[i]$), and this is in turn represented using Z and O. This already gives the result in terms of R.

To express the result in terms of H_k, we have the bound $R \le nH_k + \sigma^k$ for any k [41], which for $k \le \alpha\log_\sigma n$ is at most $nH_k + n^\alpha$, $\alpha < 1$. Let us first consider the uninteresting case $R = \Theta(n)$. The space formula $2R\log\frac{n}{R} + \Theta(R) + o(n)$ becomes simply $\Theta(n)$. Since $nH_k + n^\alpha \ge R = \Theta(n)$, it follows that $H_k = \Theta(1)$ and thus it also holds $nH_k(2\log\frac{1}{H_k} + O(1)) + o(n) = \Theta(n)$.

The interesting case is $H_k = o(1)$, and therefore $R = o(n)$. Since $2R\log\frac{n}{R}$ is an increasing function of R for $R < n/e$, we have $2R\log\frac{n}{R} + \Theta(R) \le 2(nH_k + n^\alpha)\log\frac{n}{nH_k+n^\alpha} + \Theta(nH_k + n^\alpha)$. This is at most $2nH_k\log\frac{1}{H_k} + \Theta(nH_k) + O(n^\alpha\log n)$, which is upper bounded by $2nH_k(\log\frac{1}{H_k} + \Theta(1)) + O(\frac{n\log\log n}{\log n})$. □

We emphasize that, although our somewhat crude upper bounds do not show it, our representation is asymptotically never larger than the original H, that is, at most $2n + o(n)$ bits.

Finally, we note that, given SA and LCP, the construction time of our data structure is linear. This is not the case for the original solution to compressed bitmaps [42], which need perfect hashing. However, the only essential part of their solution is the (c, o) entropy-bounded encoding which, coupled with a directory using $O(\frac{n\log\log n}{\log n})$ extra bits, can obviously be built in linear time and allows extracting any $O(\log n)$-bits block from the bitmap in constant time. On top of that, one can add the $O(\frac{n\log\log n}{\log n})$ extra bits needed to answer *rank* [39] and *select* [22] in constant time on uncompressed bitmaps, replacing their accesses to the bitmap by those to the (c, o) encoding. Those structures can also be built in linear time. Indeed one can use encodings that achieve higher-order compression of the bitmap [15], albeit our analysis does not take advantage of such a result.

4. Previous/Next Smaller Value 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 they are also of independent interest. At the end we extend our results to achieve constant time and a linear number of extra bits of space.

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: $\mathrm{NSV}_S(i) = \min\{j,\ (i < j \leq n \wedge S[j] \prec S[i]) \vee j = n+1\}$ and $\mathrm{PSV}_S(i) = \max\{j,\ (1 \leq j < i \wedge S[j] \prec S[i]) \vee j = 0\}$, respectively.

The key idea to solve these queries reminds that for *findopen* and *findclose* operations in balanced parentheses, in particular the recursive version [21]. 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 $\mathrm{NSV}(i)$ for $\mathrm{NSV}_S(i)$. We split $S[1,n]$ into consecutive *blocks* of b values. A position i will be called *near* if $\mathrm{NSV}(i)$ 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 $\mathrm{NSV}(i_1) \geq \mathrm{NSV}(i_2) \ldots \geq \mathrm{NSV}(i_s)$.

A far position i will be called a *pioneer* if $\mathrm{NSV}(i)$ is *not* in the same block of $\mathrm{NSV}(j)$, being j the largest far position preceding i (the first far position is also a pioneer). It follows that, if i is not a pioneer and j is the last pioneer preceding i, then $\mathrm{NSV}(i)$ is in the same block of $\mathrm{NSV}(j) \geq \mathrm{NSV}(i)$. Hence, to solve $\mathrm{NSV}(i)$, we find j and then scan (left to right) the block $S[\lceil \mathrm{NSV}(j)/b \rceil - b + 1 \ldots \mathrm{NSV}(j)]$, 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 [27], P can be represented using $O(\frac{n \log b}{b}) + O(\frac{n \log\log n}{\log n})$ bits of space [42]. With this representation, we can easily find the last pioneer preceding a far position i, as $j = \mathit{select}(P, \mathit{rank}(P, i))$. 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 $\mathrm{NSV}(j) = A[\mathit{rank}(P,j)]$. 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 applying the idea recursively. Instead of storing the answers explicitly in array A, we will form a (virtual) *reduced* sequence $S'[1, \leqslant 2n']$ containing all the pioneer values i and their answers $\mathrm{NSV}(i)$. Sequence S' is not explicitly stored. Rather, we set up a bitmap $R[1,n]$ where the selected values of S are marked. Hence we can retrieve any value $S'[i] = S[\mathit{select}(R,i)]$. 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 [42].

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 $i' = \mathit{rank}(R,i)$

in S', and $\mathrm{NSV}(i)$ is mapped to $j' = rank(R, \mathrm{NSV}(i))$, then $j' = \mathrm{NSV}_{S'}(i')$, because all values in $S'[i'+1 \ldots j'-1]$ correspond to values within $S[i+1 \ldots \mathrm{NSV}(i)-1]$, which by definition of NSV are not smaller than $S[i]$. Hence, we can find $\mathrm{NSV}(i)$ for pioneers i by the corresponding recursive query on S', $\mathrm{NSV}(i) = select(R, \mathrm{NSV}_{S'}(rank(R, i)))$. We are left with the problem of solving queries $\mathrm{NSV}_{S'}(i)$.

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 $S''[1, \leqslant 2n'']$, $n'' = O(n'/b) = O(n/b^2)$.

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 can be as small as half the size of R^ℓ). The sum of all the P^ℓ and R^ℓ takes order of

$$\begin{aligned} \sum_{1 \le \ell \le r} \left(\frac{n}{b^\ell} \log(b^\ell) + \frac{n \log\log n}{\log n} \right) &\le n \log b \left(\sum_{\ell \ge 1} \frac{\ell}{b^\ell} \right) + r \, \frac{n \log\log n}{\log n} \\ &= O\left(\frac{n \log b}{b} + r \, \frac{n \log\log n}{\log n} \right). \end{aligned}$$

We now state the main result of this section.

Theorem 2. *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.*

Proof. The general formula for any r, b has been obtained throughout this 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 $r \ge \frac{\log\log n + \log f(n) - O(1)}{\log b}$. Hence $rb \ge \frac{b}{\log b}(\log\log n + \log f(n) - O(1))$. 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 means of 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 $\log\log n + \log f(n) = O(\log\log n)$. □

Note that it is possible to get any time complexity of the form $\omega(\log\log n)$ while using $o(n)$ bits of space. In particular we highlight some cases of interest:

- $f = \log\log n$: with a data structure taking $O(\frac{n}{\log\log n})$ bits of space, the queries are solved in time $O(\log^2\log n)$.
- $f = \frac{\log^\epsilon n}{\log\log n}$: with a data structure taking $O(\frac{n\log\log n}{\log^\epsilon n})$ bits, for any constant $0 < \epsilon \leq 1$, the queries can be solved in time $O(\log^\epsilon n)$. The least space we can use corresponds to $\epsilon = 1$, where we reach logarithmic time.
- $f = \log^* n$: with a data structure taking $O(\frac{n}{\log^* n})$ bits of space (still $o(n)$ but pushing it to the extreme), we achieve time $O(\log^* n \log\log n)$.

As for construction time, we note that each level of the structure can be built in time proportional to the length of its array: We traverse the sequence and keep in a stack the positions whose NSV has not yet been found. Each new integer pops those larger than it before pushing itself in the stack. The last far position popped before the end of each block is a pioneer. As the recursive structure considers exponentially decreasing array sizes (n/b^ℓ), all the sizes add up to $O(n)$. However, we must build compressed bitmaps P^ℓ and R^ℓ. Each can be built in time linear on its *nominal* size, which is always n, and therefore the total construction time amounts to $O(rn)$. According to our previous computations, this is $O(n(1 + \frac{\log\log n}{\log f(n)}))$, which can be as bad as $O(n\log\log n)$.

4.1. Achieving Constant Time with Linear Space

If we use $\Theta(n)$ bits of space in Theorem 2, the time is still $O(\log\log n)$. However, we can do better within that space.

Theorem 3. *Queries* PSV *and* NSV *can be solved in constant time by using* $4n + o(n)$ *bits of space.*

Proof. We reduce PSV and NSV queries to $O(1)$ *findopen* and *findclose* operations in balanced parentheses [21]. For NSV, for $1 \leq i \leq n+1$ in this order, write a '(' and then x ')'s if there are x cells $S[j]$ for which $\mathrm{NSV}(j) = i$. The resulting sequence B is balanced if a final ')' is appended, and $\mathrm{NSV}(i)$ 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. Range Minimum Queries in Sublinear Space

In this section we show how to preprocess a sequence $S[1, n]$ of arbitrary symbols (which can be compared with $\preceq$) such that RMQ_S can be answered in sublogarithmic time, using $o(n)$ bits of additional space.

Definition 3. Let $S[1,n]$ be a sequence of elements drawn from a set with a total order $\preceq$. The *range minimum query* is defined as follows: $\mathrm{RMQ}_S(i,j) = \mathrm{argmin}_{i\le\ell\le j} S[\ell]$, where argmin refers to order $\preceq$.

A well-known strategy [17, 45] divides S 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 RMQ_S 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 $O(\frac{n}{b_i}\log(\frac{b_{i-1}}{b_i})\log(b_{i-1})) = O(\frac{n}{b_i}\log^2(b_{i-1}))$ 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 all $j \le \lfloor\log(\frac{b_{i-1}}{b_i})\rfloor$ (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. Queries q_1 and q_{2r+1} are solved by scanning in time $O(b_r)$, and all other queries can be answered by table-lookups in total time $O(r)$.[4] The final answer is obtained by comparing (using $\preceq$) at most $2r+1$ minima.

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

Lemma 2. *Having constant-time access to elements in $S[1,n]$ and to compute operation $\preceq$, it is possible to answer range minimum queries on S in time $O(f(n)\log^2 f(n))$ using $O(\frac{n}{f(n)})$ bits of space, for any $f(n) = \Omega(\log^{[r]} n)$ and any constant $r \ge 0$. (By $\log^{[r]} n$ we denote r applications of $\log$ to n.)*

Proof. 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 $1 \le i \le r+1$, 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 $b_{r+1} = \Theta(f(n)\log^2 b_r) = \Theta(f(n)(\log f(n) + \log^{[r+1]} n)^2) = \Theta(f(n)\log^2 f(n))$. □

Some interesting tradeoffs follow:

- $f = \log\log n$: with $O(\frac{n}{\log\log n})$ bits of space, we answer queries in time $O(\log\log n \cdot \log^2\log\log n)$.
- $f = \frac{\log^\epsilon n}{\log\log n}$: with $O(\frac{n\log\log n}{\log^\epsilon n})$ bits of space, for any constant $0 < \epsilon \le 1$, the query time is $O(\log^\epsilon n\,\log\log n)$.
- $f = \log^* n$: although not directly covered by the lemma, it is not hard to see that one can get $O((\log^* n\log\log^* n)^2)$ time and $O(\frac{n}{\log^* n})$ bits of space, by choosing $r = (\log^* n) - 1$ and $f = (\log^* n)^2$.

[4] The constant-time solutions [45, 17] also solve q_1 and q_{2r+1} by accessing tables that require $\Theta(n)$ bits.

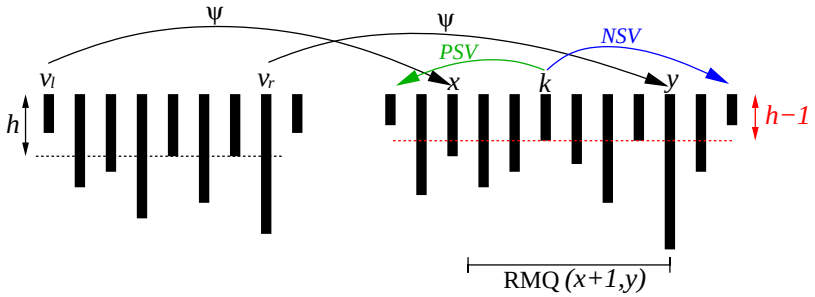

Figure 1: Left: Illustration to the representation of suffix tree nodes. The lengths of the bars indicate the LCP values. All leaves in the subtree rooted at $v = [v_l, v_r]$ share a longest common prefix of length at least h. Right: Schematic view of the SLink operation. From v, first follow ψ, then 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.

Construction time for this structure is $O(n)$. This is dominated by the time to scan the array in order to fill the last level, $r+1$. All the rest is filled using dynamic programming in constant time per cell stored. These add up to $O(\frac{n}{b_i} \log b_{i-1}) = O(\frac{n}{f(n)})$ per level, and the total number of levels is $r + 1 = O(1)$.

6. 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, 9, 43], 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 $\mathsf{LCP}[v_l] < h$, $\mathsf{LCP}[i] \geq h$ for all $v_l < i \leq v_r$, and $\mathsf{LCP}[v_r + 1] < h$. Fig. 1 (left) illustrates. We state the easy yet fundamental

Lemma 3. *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 $k = \mathrm{RMQ}_{\mathsf{LCP}}(v_l+1, v_r)$.*

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

Now we have all the ingredients for navigating in the suffix tree. The operations are described in the following. We will make use of RMQ and PSV/NSV queries on the array LCP. 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 PSV/NSV queries. As these are carried out on LCP, and their dominating cost is the number of access to the array, their times will be multiplied by t_{LCP}, the cost to access any LCP cell.

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

SDepth(v)/Locate(v): According to Lemma 3, SDepth(v) can be computed in time $O(t_{\mathrm{RMQ}} \cdot t_{\mathsf{LCP}})$ 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 [43]. 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 $[\mathrm{PSV}(k), \mathrm{NSV}(k)-1]$. Time is $O(t_{\mathrm{PNSV}} \cdot t_{\mathsf{LCP}})$.

FChild(v): If v is a leaf, return null. Otherwise, because the minima in $[v_l, v_r]$ are v's h-indices [17], the first child of v is given by $[v_l, \mathrm{RMQ}(v_l+1, v_r)-1]$, assuming that RMQs always return the leftmost minimum in the case of ties (which is easy to arrange). Time is $O(t_{\mathrm{RMQ}} \cdot t_{\mathsf{LCP}})$.

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 $[v_r+1, \mathrm{RMQ}(v_r+2, w_r)-1]$. The overall time is $O((t_{\mathrm{RMQ}} + t_{\mathrm{PNSV}}) \cdot t_{\mathsf{LCP}})$.

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 $k = \mathrm{RMQ}(x+1, y)$ (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 $[\psi(v_l), \psi(v_r)]$; see also [1, Lemma 7.5]). The final result is then given by $[\mathrm{PSV}(k), \mathrm{NSV}(k)-1]$. Time is $O(t_\psi + (t_{\mathrm{PNSV}} + t_{\mathrm{RMQ}}) \cdot t_{\mathsf{LCP}}))$. 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 $\psi^i(v) = \mathsf{SA}^{-1}[\mathsf{SA}[v]+i]$ [43]. Time is thus $O(t_{\mathsf{SA}} + (t_{\mathrm{PNSV}} + t_{\mathrm{RMQ}}) \cdot t_{\mathsf{LCP}})$.

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 [45]: $k = \mathrm{RMQ}(v_r+1, w_l)$. The final answer is again $[\mathrm{PSV}(k), \mathrm{NSV}(k)-1]$. Time is $O((t_{\mathrm{RMQ}} + t_{\mathrm{PNSV}}) \cdot t_{\mathsf{LCP}})$.

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 $T_{\mathsf{SA}[p]+\mathsf{LCP}[p]} = \mathrm{Letter}([p,p], \mathsf{LCP}[p]+1) = a$. Then the final result is given by $[p, \mathrm{RMQ}(p+1, v_r)-1]$, 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 Section 5). Intervals $[v_l, x-1]$ and $[y, v_r]$ can be scanned for p in time $O(t_{\mathrm{RMQ}} \cdot (t_{\mathsf{LCP}} + t_{\mathsf{SA}}))$. 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 [45] (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 [18, Section 3.2]. Overall time is $O((t_{\mathsf{LCP}} + t_{\mathsf{SA}}) \cdot t_{\mathrm{RMQ}} + \log \sigma \cdot t_{\mathsf{SA}})$.

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 $v = [v_l, v_r]$, LETTER$(v, 1) = L[rank(C, v_l)]$. L requires $O(\sigma \log \sigma)$ bits and C, represented as a compressed bitmap [42], 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 LETTER(v, i) = 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 [24, 44, 14].

TDEPTH(v): Tree-depth can be maintained while performing some traversal operations such as ROOT, FCHILD, NSIBLING, CHILD, PARENT, LAQT. However, other operations are not easily handled.

Yet, there is also a direct way to support TDEPTH, using other $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits of space. The idea is similar to Sadakane's representation of LCP [45]: 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 $k + \mathsf{TDE}[\psi^k(\mathsf{SA}^{-1}[1])]$, for $0 \le k < n$, 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 H (Section 3), so the resulting sequence can be compressed to $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits using the same mechanism as for LCP. The time is thus $O(t_{\mathrm{RMQ}} \cdot t_{\mathsf{LCP}})$, just as for SDEPTH. Leaves can be solved in $O(t_{\mathsf{SA}})$ time by TDEPTH$(v) = 1 + \max(\mathsf{TDE}[\mathsf{SA}[v]], \mathsf{TDE}[\mathsf{SA}[v+1]])$.

LAQS(v, d): Let $u = [u_l, u_r] =$ 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 $k = \mathrm{RMQ}(\lfloor v_l/2 \rfloor, v_l)$, we check if $\mathsf{LCP}[k] \ge d$. 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 $O(\log n \cdot t_{\mathrm{RMQ}} \cdot t_{\mathsf{LCP}})$.

LAQT(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 $O(\log n \cdot t_{\mathrm{RMQ}} \cdot t_{\mathsf{LCP}})$.

7. 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) [44], 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) [14] gives the best space, but our CST over AFFM is worse than Russo et al.'s CST (RCST) [43] both in time and space. Instead, we focus on using Grossi et al.'s compressed suffix array (GCSA) [24], 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) [45] 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 $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ extra bits. Our CST times are $t_{\mathsf{LCP}} = t_{\mathsf{SA}}$ (recall Section 3), 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, σ, the nature of the $o(n)$ extra bits in our CST, 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 Theorem 2 and Lemma 2). Thus, our times are $t_{\mathrm{RMQ}} = \log^{\epsilon'} n \log^2 \log n$ and $t_{\mathrm{PNSV}} = \log^{\epsilon'} n \log\log n$. RCST's γ and δ are $O(\log^{1+\epsilon'} n)$. Table 1 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 space used in the entropy-related part will be 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 can achieve sublogarithmic times (except for level ancestor queries), being the only compressed suffix tree achieving it within $o(n)$ extra space.

We have assumed $\sigma = o(n)$, so that $\sigma \log n = o(n \log \sigma)$. Yet, we remark that our scheme is not so attractive on large alphabets. If $\sigma = \Theta(n^\beta)$ for constant β, then our extra space includes a term $\Theta(n \log\log n)$, just as in SCST over SCSA, while the latter is clearly faster.

Both the suffix array SA and the longest common prefix sequence LCP can be built in linear time [31, 32, 29, 30]. From it, arrays SA^{-1} and Ψ are easily built in linear time. The compressed suffix array GCSA, on the other hand, needs $O(n \log \sigma)$ construction time [24]. Most of the data structures we have added are built in $O(n)$ time, except for the $O(rn)$ of Theorem 2. This can be as high as $O(n \log\log n)$, but not for our CST: As there is already a cost of the form $\Omega(\log^\epsilon n)$ per operation coming from GCSA, there is no point in using $f(n)$ smaller than $\log^{\epsilon'} n$ as in our comparison. For this setting we have $O(rn) = O(n/\epsilon') = O(n)$. Overall, we achieve our results with a construction time bounded by $O(n \log \sigma)$. This cost depends on GCSA, and it is also needed to build the other k-th order compressed suffix array (AFFM) [14].

Operation	Our suffix tree		Other suffix trees	
	General	over GCSA [24]	SCST [45]	RCST [43]
Space		$nH_k(2\log\frac{1}{H_k}+\frac{1}{\epsilon}+O(1))+o(n)$	$(1+\frac{1}{\epsilon})nH_k+6n+o(n)$	$nH_k+o(n)$
Root,Count,Ancestor	1	1	1	1
Locate	t_{SA}	$\log^{\epsilon} n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
SDepth	$t_{\mathsf{SA}} \cdot t_{\mathrm{RMQ}}$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
Parent	$t_{\mathsf{SA}} \cdot t_{\mathrm{PNSV}}$	$\log^{\epsilon+\epsilon'} n \log\log n$	1	$\log^{1+\epsilon'} n$
FChild	$t_{\mathsf{SA}} \cdot t_{\mathrm{RMQ}}$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{1+\epsilon'} n$
NSibling	$t_{\mathsf{SA}}(t_{\mathrm{RMQ}}+t_{\mathrm{PNSV}})$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{1+\epsilon'} n$
SLink,LCA	$t_{\mathsf{SA}}(t_{\mathrm{RMQ}}+t_{\mathrm{PNSV}})$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{1+\epsilon'} n$
SLink,LCA	$t_{\mathsf{SA}}(t_{\mathrm{RMQ}}+t_{\mathrm{PNSV}})$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{1+\epsilon'} n$
SLink^i	$t_{\mathsf{SA}}(t_{\mathrm{RMQ}}+t_{\mathrm{PNSV}})$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
Child	$t_{\mathsf{SA}}(t_{\mathrm{RMQ}}+\log\sigma)$	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n \log\log n$
Letter	t_{SA}	$\log^{\epsilon} n$	$\log^{\epsilon} n$	$\log^{1+\epsilon'} n$
TDepth	$t_{\mathsf{SA}} \cdot t_{\mathrm{RMQ}}$ (*)	$\log^{\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{2+2\epsilon'} n$
LAQs	$t_{\mathsf{SA}} \cdot t_{\mathrm{RMQ}} \cdot \log n$	$\log^{1+\epsilon+\epsilon'} n \log^2 \log n$	Not supported	$\log^{1+\epsilon'} n$
LAQt	$t_{\mathsf{SA}} \cdot t_{\mathrm{RMQ}} \cdot \log n$ (*)	$\log^{1+\epsilon+\epsilon'} n \log^2 \log n$	1	$\log^{2+2\epsilon'} n$

(*) Our CST needs other $nH_k(2\log\frac{1}{H_k}+O(1))+o(n)$ 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 $o(n)$ space is $O(\frac{n}{\log^{\epsilon'} n})$. SCST is assumed to run over GCSA.

Construction space is $O(n \log n)$ bits, and reducing it is a challenging issue for future work, as well as handling dynamic text collections (there exists some work on this track for SCST [7]). Others are related to achieving improved results on our basic data structures of Theorems 2/3 and Lemma 2, or proving this is not possible. Finally, we leave open the challenge of achieving sublogarithmic time for the suffix tree operations while using optimal space, $nH_k + o(n \log \sigma)$ bits.

8. Towards a Practical Implementation

A practice-oriented future work direction is to implement our proposal. A first concern is how to choose a practical compressed full-text index, so that ψ, SA, SA^{-1}, and so on, are well supported. This topic has been studied in depth [13], and there exist several alternatives known to work well in practice.

When considering the suffix-tree-specific part, a crucial point, where a practical implementation might have to differ from the theoretical proposal, is in the representation of LCP information. The time to access LCP array is the key to the overall performance, and if implemented verbatim from the theoretical proposal, it is likely to be high.

An interesting practical alternative, for which however we have not been able to find sufficiently good theoretical space guarantees, is to represent LCP array more directly, in a way that offers fast access and at the same time compressed space.

8.1. Re-Pair Based Compression

Re-Pair [35] is a grammar-based compression method, which proceeds as follows: (1) Find the most repeated pair ab in the sequence; (2) Replace all its occurrences by a new symbol s; (3) Add a rule $s \to ab$ to a dictionary; (4) Iterate until every pair is unique.

The reason to choose Re-Pair as a compression method is not arbitrary. In recent work [23], Re-Pair is used to compress the differentially encoded suffix array, $\mathsf{SA}'[i] = \mathsf{SA}[i] - \mathsf{SA}[i-1]$. It is shown that Re-Pair achieves $O(R \log \frac{n}{R} \log n)$ bits of space, being R the number of runs in ψ. The rationale is that, except on the first cell of each run, it holds $\mathsf{SA}'[i]\mathsf{SA}'[i+1] = \mathsf{SA}'[\psi(i)]\mathsf{SA}'[\psi(i)+1]$, and therefore there are at most $2R$ cells in SA' where this does not hold. From that it can be shown that Re-Pair will compress SA' to at most $8R \log \frac{n}{4R} + O(R)$ integers (counting both dictionary and sequence [23]).

Now, within a run in ψ (except for the first cell) it also holds that $\mathsf{LCP}[\psi(i)] = \mathsf{LCP}[i] - 1$, and thus a differential encoding $\mathsf{LCP}'[i] = \mathsf{LCP}[i] - \mathsf{LCP}[i-1]$ also satisfies $\mathsf{LCP}'[i]\mathsf{LCP}'[i+1] = \mathsf{LCP}'[\psi(i)]\mathsf{LCP}'[\psi(i)+1]$ within runs (this is related to our development in Section 3). Thus the analysis done for SA' applies verbatim, showing that Re-Pair would compress LCP' to $O(R \log \frac{n}{R} \log n)$ bits. Considering that $R \leq nH_k + \sigma^k$ [36], we get a (crude) upper bound of $O(nH_k \log \frac{1}{H_k} \log n)$ bits. This is worse by an $O(\log n)$ factor than our result in Section 3, yet still goes to zero when $H_k \to 0$, and may offer much faster access to individual values, by adding some extra data structures that use negligible extra space [23]. Some experimental results on access time to suffix arrays are given in the same article. Although the time is not constant, it is very low in practice and enjoys a local access pattern.

A brief experiment to show the potential of this idea was carried out on the 50MB English collection of site *Pizza&Chili*[5]. Including the necessary structures to provide

[5] `http://pizzachili.dcc.uchile.cl` or `http://pizzachili.di.unipi.it`.

random access to the LCP array, Re-Pair compressed it to less than 49MB, that is, almost the same size of the original text (and 25% the size of a plain integer representation).

The other structures to complete the suffix tree are a compressed suffix array (which in practice takes 30%-60% of the original text, and replaces it [13]) and other structures that can be made as small as desired, in exchange for time. We anticipate that, within 2 times the space of the original text (so that the text can be discarded), we could have a fully-functional compressed suffix tree, which would operate rather efficiently.

What is also intriguing is that the differential LCP', despite this entropy bound, could compress *better* when the text is *less* compressible. An incompressible text yields a balanced suffix tree, where most of the nodes are at depth near $\log_\sigma n$ [47]. Hence the differences between consecutive LCP values tend to be small, and the differential sequence becomes more compressible.

8.2. Entropy-Bounded Sequence Compression

An alternative direct representation of LCP achieves entropy bounds *on sequence* LCP*, not on the original text*, and as such does not offer a theoretical space guarantee comparable to those seen in Section 3. In exchange, it achieves constant access time to LCP in theory.

The idea is to encode LCP using a recent compressed sequence representation [15], which applied on a sequence S over alphabet σ achieves $|S|H_k(S) + o(|S|\log\sigma)$ bits of space, for any $k = o(\log_\sigma |S|)$. This idea entails a significant overhead in the sublinear part. However, this compression method has already been tested successfully on preprocessing schemes for $O(1)$-RMQs on LCP-arrays [19], though not on LCP-arrays themselves.

A more serious problem, in our case, is that $\sigma = |S| = \Theta(n)$, and thus the result gives us just $nH_0(\mathsf{LCP}) + o(n\log n)$ bits of space. For the sake of compressing to zero-order entropy, one could equivalently represent LCP using a wavelet tree designed for large alphabets [8], and achieve very little sublinear overhead, albeit access time raises to $O(\log\sigma) = O(\log n)$.

8.3. Reducing Sublinear Terms

In practice, the terms of the form $o(n)$ can be significant and should be considered carefully. One alternative to exchange time for sublinear space would be to consider the relationship between PSV/NSV and RMQ queries, so as to spend space for one of them and answer the other in terms of the first.

For example, $\mathrm{RMQ}_S(i,j)$ can be solved by iteratively applying $i \leftarrow \mathrm{NSV}_S(i)$ until $i > j$, then the answer is the previous i value. This could replace the scanning of the smallest RMQ blocks, and this faster scanning could allow us using larger blocks in practice. Alternatively, $j \leftarrow \mathrm{NSV}_S(i)$ could be solved by applying $\mathrm{RMQ}_S(i, i+2^k)$ for increasing k until finding a value smaller than $S[i]$, and then binary searching for the first $j \in [i+2^{k-1}, i+2^k]$ such that $S[\mathrm{RMQ}_S(i,j)] \prec S[i]$. PSV could obviously be dealt with similarly, and polylogarithmic time guarantees would be maintained.

Other intriguing possibilities are, for example, implementing the RMQ mechanism over the Re-Pair phrases instead of over fixed-sized blocks, as then the dictionary could factor out repeated blocks across the text.

Acknowledgments.. We thank Rodrigo González for the experimental result shown in Section 8.1. JF wishes to thank Volker Heun and Enno Ohlebusch for interesting discussions on this subject.

References

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proc. 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 258–264, 2002.

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

[4] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[5] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

[6] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[7] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.

[8] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[9] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Proc. 33rd International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 4051, pages 358–369, 2006.

[10] M. Crochemore, C. Iliopoulos, M. Kubica, M. Sohel Rahman, and T. Walen. Improved algorithms for the range next value problem and applications. In *Proc. 25th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 205–216, 2008.

[11] M. Crochemore, C. Iliopoulos, and M. Sohel Rahman. Finding patterns in given intervals. In *Proc. 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, LNCS 4708, pages 645–656, 2007.

[12] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. 5th Workshop on Experimental Algorithmics (WEA)*, LNCS 4007, pages 134–145, 2006.

[13] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009.

[14] 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.

[15] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 690–696, 2007.

[16] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 36–48, 2006.

[17] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, LNCS 4614, pages 459–470, 2007.

[18] J. Fischer and V. Heun. Range median of minima queries, Super-Cartesian trees, and text indexing. In *Proc. 19th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 239–252, 2008.

[19] J. Fischer, V. Heun, and H. Stühler. Practical entropy-bounded schemes for $O(1)$-range minimum queries. In *Proc. 18th Data Compression Conference (DCC)*, pages 272–281, 2008.

[20] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165, 2008.

[21] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368:231–246, 2006.

[22] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4051, pages 370–381, 2006.

[23] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[24] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[25] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[26] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[27] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[28] J. Kärkkäinen and S. Rao. *Algorithms for Memory Hierarchies*, chapter 7: Full-text indexes in external memory, pages 149–170. LNCS 2625. Springer, 2003.

[29] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 943–955, 2003.

[30] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2089, pages 181–192, 2001.

[31] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2676, pages 186–199, 2003.

[32] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2676, pages 200–210, 2003.

[33] P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 184–194, 2007.

[34] S. Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[35] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.

[36] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[37] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

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

[39] I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[40] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

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

[42] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[43] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 362–373, 2008.

[44] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

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

[46] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal Discrete Algorithms*, 5(1):12–22, 2007.

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