Sparse Suffix and LCP Array: Simple, Direct, Small, and Fast*

Lorraine A. K. Ayad[1], Grigorios Loukides[2], Solon P. Pissis[3,4], and Hilde Verbeek[3]

[1] Brunel University London, London, UK
lorraine.ayad@brunel.ac.uk
[2] King's College London, London, UK
grigorios.loukides@kcl.ac.uk
[3] CWI, Amsterdam, the Netherlands
{solon.pissis,hilde.verbeek}@cwi.nl
[4] Vrije Universiteit, Amsterdam, the Netherlands

Abstract. Sparse suffix sorting is the problem of sorting $b = o(n)$ suffixes of a string of length n. Efficient sparse suffix sorting algorithms have existed for more than a decade. Despite the multitude of works and their justified claims for applications in text indexing, the existing algorithms have not been employed by practitioners. Arguably this is because there are no simple, direct, *and* efficient algorithms for sparse suffix array construction. We provide two new algorithms for constructing the sparse suffix and LCP arrays that are simultaneously simple, direct, small, and fast. In particular, our algorithms are: *simple* in the sense that they can be implemented using only basic data structures; *direct* in the sense that the output arrays are not a byproduct of constructing the sparse suffix tree or an LCE data structure; *fast* in the sense that they run in $\mathcal{O}(n \log b)$ time, in the worst case, or in $\mathcal{O}(n)$ time, when the total number of suffixes with an LCP value greater than $2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ is in $\mathcal{O}(b/\log b)$, matching the time of optimal yet much more complicated algorithms [Gawrychowski and Kociumaka, SODA 2017; Birenzwige et al., SODA 2020]; and *small* in the sense that they can be implemented using *only* $8b + o(b)$ machine words. We also show that our second algorithm can be trivially amended to work in $\mathcal{O}(n)$ time for any uniformly random string. Our algorithms are non-trivial space-efficient adaptations of the Monte Carlo algorithm by I et al. for constructing the sparse suffix tree in $\mathcal{O}(n \log b)$ time [STACS 2014].

Keywords: suffix array · LCP array · suffix sorting · sparse suffix sorting.

* SPP and HV are supported by the PANGAIA project (GA 872539). SPP is supported by the ALPACA project (GA 956229). HV is supported by a Constance van Eeden Fellowship.

1 Introduction

Let $T = T[1 \,..\, n]$ be a string of length n over an ordered alphabet Σ. Further let $\mathcal{B} \subseteq [1, n]$ be a set of $b > 1$ positions in T. *Sparse suffix sorting* is the problem of sorting the set of suffixes $T_{\mathcal{B}} = \{T[i \,..\, n] : i \in \mathcal{B}\}$ lexicographically [18]. This is achieved by constructing the sparse suffix array. The *sparse suffix array* $\mathsf{SSA} = \mathsf{SSA}[1 \,..\, b]$ is the array containing the positions in $\mathcal{B}$ in the lexicographical order of the suffixes in $T_{\mathcal{B}}$. The associated *sparse longest common prefix array* $\mathsf{SLCP} = \mathsf{SLCP}[1 \,..\, b]$ stores the length $\mathsf{SLCP}[i]$ of the longest common prefix of $T[\mathsf{SSA}[i-1] \,..\, n]$ and $T[\mathsf{SSA}[i] \,..\, n]$ when $i \in [2, n]$ or 0 when $i = 1$. The SSA and SLCP array can be used to construct the sparse suffix tree in linear time using the algorithm by Kasai et al. [20]. The *sparse suffix tree* is the compacted trie of the set $T_{\mathcal{B}}$. Vice-versa, the SSA and SLCP array can be obtained in linear time via a pre-order traversal of the sparse suffix tree.

Sparse suffix sorting was introduced as a fundamental step in the construction of compressed or sparse text indexes [18]. Modern compressed text indexes [24, 10], practical indexes for long patterns [15, 22, 23, 2], and sublinear-space string algorithms [3, 5] rely on sparse suffix sorting: they first sample a sublinear number of "important" suffixes, which they next sort to construct their final solution. Efficient sparse suffix sorting algorithms have existed for more than a decade. The following algorithms construct SSA explicitly, or implicitly by first constructing the sparse suffix tree. Since the size of SSA (and the size of sparse suffix tree) is $\Theta(b)$, the goal of these algorithms is to use $\mathcal{O}(b)$ words of space assuming read-only random access to T. Kärkkäinen et al. presented a deterministic $\mathcal{O}(n^2/s)$-time and $\mathcal{O}(s)$-space algorithm, for any $s \in [b, n]$ [17, Section 8]. Bille et al. presented a Monte Carlo $\mathcal{O}(n \log^2 b)$-time and $\mathcal{O}(b)$-space algorithm [6], as well as a Las Vegas $\mathcal{O}(n \log^2 n + b^2 \log b)$-time and $\mathcal{O}(b)$-space algorithm. I et al. presented a Monte Carlo $\mathcal{O}(n + (bn/s) \log s)$-time and $\mathcal{O}(s)$-space algorithm, for any $s \in [b, n]$ [16] and a Las Vegas $\mathcal{O}(n \log b)$-time and $\mathcal{O}(b)$-space algorithm. Gawrychowski and Kociumaka [14] presented a Monte Carlo $\mathcal{O}(n)$-time and $\mathcal{O}(b)$-space algorithm and a Las Vegas $\mathcal{O}(n\sqrt{\log b})$-time and $\mathcal{O}(b)$-space algorithm. Birenzwige et al. [7] presented a Las Vegas algorithm running in $\mathcal{O}(n)$ time using $\mathcal{O}(b)$ space. Besides this they also presented a deterministic $\mathcal{O}(n \log \frac{n}{b})$-time and $\mathcal{O}(b)$-space algorithm, for any $b = \Omega(\log n)$.

The following algorithms also construct SSA, but they work in the *restore model* [9]: an algorithm is allowed to overwrite parts of the input, as long as it can restore it to its original form at termination. Fischer et al. [12] presented a deterministic $\mathcal{O}(c\sqrt{\log n} + b \log b \log n \log^* n)$-time and $\mathcal{O}(b)$-space algorithm, where c is the number of letters that must be compared for distinguishing the suffixes in $T_{\mathcal{B}}$. In some cases, this runs in sublinear extra time; extra refers to the linear cost of loading T in memory. Prezza [26] presented a Monte Carlo $\mathcal{O}(n + b \log^2 n)$-time algorithm using $\mathcal{O}(1)$ words of space.

Motivation. Despite the multitude of works on sparse suffix sorting and their justified claims for applications in text indexing, the existing algorithms have not been employed by practitioners. Arguably this is because there are no simple, direct, *and* efficient algorithms for SSA construction. The $\mathcal{O}(n)$-time algorithms of Gawrychowski and Kociumaka [14] and of Birenzwige et al. [7] are far from simple and do not seem to be practically promising either. The former (Monte Carlo) algorithm relies heavily on the construction of compacted tries, which induce high constants in space usage, and on a recursive application of difference cover to construct a Longest Common Extension (LCE) data structure. The latter (Las Vegas) algorithm relies on an intricate partitioning scheme (sampling) to construct SSA and on an LCE data structure to compute the SLCP array. The Monte Carlo $\mathcal{O}(n \log b)$-time algorithm of I et al. [16] is simple but it also relies heavily on compacted tries, which makes it less likely to be employed by practitioners for SSA construction. The Monte Carlo $\mathcal{O}(n + b \log^2 n)$-time algorithm of Prezza [26] makes heavy usage of an LCE data structure as well: constructing the SSA and SLCP array is a byproduct of an in-place LCE data structure. The latter algorithm is, to the best of our knowledge, the only algorithm which has been implemented (at least in a simplified form). Due to the interest in sparse suffix sorting and the above characteristics of the existing algorithms, we were motivated to revisit this problem to develop efficient, yet simple and direct, algorithms for SSA construction. Such algorithms may serve as baselines for practitioners to engineer the SSA and SLCP array construction.

Our Model and Results. We assume the standard word RAM model with word size $\Theta(\log n)$; basic arithmetic and bit-wise operations on $\mathcal{O}(\log n)$-bit integers take $\mathcal{O}(1)$ time. We assume that we have a read-only random access string T of length n over an integer alphabet $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$, a read-only integer array A of size b storing the b elements of $\mathcal{B}$, and two write-only integer arrays SSA and SLCP, each of size b. We thus count the amount of *extra space in machine words* used to construct the SSA and SLCP array. We present two algorithms:

1. Our first algorithm, Main-Algo, constructs SSA and SLCP *directly*; i.e., without first explicitly constructing the sparse suffix tree or an LCE data structure (see Section 3). Its time complexity is $\mathcal{O}(n + (bn/s) \log s)$ and its space complexity is $s + 7b + o(b)$ machine words, for any chosen $s \in [b, n]$. It is a Monte Carlo algorithm that returns the correct output *with high probability*; i.e., with probability at least $1 - n^{-c}$, for any constant $c \geq 1$ chosen at construction time. Main-Algo is *simple* in the sense that it can be implemented using only *basic data structures* (e.g., dictionaries and arrays) readily available in widely-used programming languages (e.g., `C++`, `Java`, or `Python`). Main-Algo is a non-trivial space-efficient simulation of the algorithm by I et al. for sparse suffix tree construction [16]. A disadvantage of these two algorithms is that they attain the $\Theta(n \log b)$ time bound for $s = b$ *in any case.* To address this, we develop Parameterized-Algo, a parameterized algorithm which is input-sensitive.
2. Our second algorithm, Parameterized-Algo, also constructs SSA and SLCP directly (see Section 4). Its time complexity is $\mathcal{O}(n + (b'n/b) \log b)$ and its space complexity is $8b + 4b' + o(b)$ machine words, where b' is the total number of suffixes $\mathsf{SSA}[i] \in \mathcal{B}$ with $\mathsf{SLCP}[i] \geq \ell$ or $\mathsf{SLCP}[i+1] \geq \ell$, where

$\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$. When $b' = \mathcal{O}(b/\log b)$, Parameterized-Algo runs in $\mathcal{O}(n)$ time, thus matching the time of the optimal yet much more complicated algorithms in [14, 7], using *only* $8b + o(b)$ machine words. It is a Monte Carlo algorithm that returns the correct output with high probability. It is *remarkably simple* as it consists of two calls of Main-Algo and a linear-time step that merges the partial results (however, the proof of correctness requires some work). The running time of Parameterized-Algo is good in the following sense: if the instance is reasonably sparse, then ℓ is large and likely $b' = \mathcal{O}(b/\log b)$, thus it runs in $\mathcal{O}(n)$ time. In any case, it runs in $\mathcal{O}(n \log b)$ time. For instance, for the full human genome (v. GRCh38) as T, where $n \approx 3 \cdot 10^9$, and for $b = \lfloor \sqrt{n} \rfloor = 56137$ suffixes selected uniformly at random, $b' = 2525 < \lfloor b/\log b \rfloor = 3558$. We also analyze the time complexity of Parameterized-Algo on random strings and show that it works in $\mathcal{O}(n)$ time (after a trivial amendment), for any string chosen uniformly at random from Σ^n and any set $T_{\mathcal{B}}$ of b suffixes of T, with high probability.

2 Preliminaries

We consider strings over an integer alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. The elements of Σ are called *letters*. A *string* $T = T[1 \mathinner{.\,.} n]$ is a sequence of letters from Σ; we denote by $|T| = n$ the *length* of T. The fragment $T[i \mathinner{.\,.} j]$ of T is an *occurrence* of the underlying *substring* $P = T[i] \ldots T[j]$ occurring at *position* i in T. A *prefix* of T is a substring of T of the form $T[1 \mathinner{.\,.} j]$ and a *suffix* of T is a substring of T of the form $T[i \mathinner{.\,.} n]$.

Karp-Rabin Fingerprints. Let T be a string of length n over an integer alphabet. Let p be a prime and choose $r \in [0, p-1]$ uniformly at random. The Karp-Rabin (KR) fingerprint [19] of $T[i \mathinner{.\,.} j]$ is: $\phi_T(i, j) = (\sum_{k=i}^{j} T[k] r^{j-k} \mod p, r^{j-i+1} \mod p)$. Clearly, if $T[i \mathinner{.\,.} i + \ell] = T[j \mathinner{.\,.} j + \ell]$ then $\phi_T(i, i + \ell) = \phi_T(j, j + \ell)$. On the other hand, if $T[i \mathinner{.\,.} i + \ell] \neq T[j \mathinner{.\,.} j + \ell]$ then $\phi_T(i, i + \ell) \neq \phi_T(j, j + \ell)$ with probability at least $1 - \ell/p$ [11]. Since we are comparing only substrings of equal length, the number of different possible substring comparisons is less than n^3. Thus, for any constant $c \geq 1$, we can set p to be a prime larger than $\max(|\Sigma|, n^{c+3})$ to make the KR fingerprint function perfect with probability at least $1 - n^{-c}$. Any KR fingerprint or p fit in one machine word of size $\Theta(\log n)$.

Lemma 1 ([16]). *Any string $T \in \Sigma^n$ can be preprocessed in $\mathcal{O}(n)$ time using $s + \mathcal{O}(1)$ machine words, for any $s \in [1, n]$, so that the KR fingerprint of any length-k fragment of T is computed in $\mathcal{O}(\min\{k, n/s\})$ time.*[5]

I et al. [16] employ the *distribute-and-collect* technique [25] to group b suffixes, according to a fixed-length common prefix by using their KR fingerprints, in $\mathcal{O}(b \log_s n)$ time. We instead use hashing to achieve the same result in $\mathcal{O}(b)$ time with high probability. This gives improved running times in some special regimes (see Theorem 2 and Theorem 3).

[5] I et al. [16] claim $\mathcal{O}(s)$ space but from their construction it is evident that in fact $s + \mathcal{O}(1)$ machine words are used.

3 Main Algorithm

Overview. A summary of our main algorithm (Main-Algo) follows with references to the pseudocode given in Algorithms 1 and 2.[6] It takes as input a string T from Σ^n and an array A of b elements, indicating the starting positions of the suffixes to be sorted. It also takes an integer j_{start}, which defines the number of iterations. In this section, j_{start} is set to $\lfloor \log n \rfloor$ (the default value), but a different value is used for the parameterized algorithm presented in Section 4.

Algorithm 1 Main-Algo

```
Input: string $T \in \Sigma^n$, integer $b$, array $A$ of $b$ integers,
and integer $j_{\text{start}}$ (default $\lfloor \log n \rfloor$)
Output: SSA and SLCP
 1: $m \leftarrow b + 1$
 2: $L_m \leftarrow (1, \ldots, b)$
 3: $B \leftarrow \{(m, 0, L_m)\}$
 4: $A[m] \leftarrow A[1]$
 5: for $j = j_{\text{start}}, \ldots, 0$ do
 6:     $B' \leftarrow \emptyset$
 7:     for $(i, k, L_i) \in B$ do
 8:         $H_i \leftarrow$ empty hash table
 9:         $s \leftarrow |L_i|$
10:         for $l \in L_i$ do
11:             $h \leftarrow \phi_T(A[l] + k, A[l] + k + 2^j - 1)$
12:             $H_i[h]$.append($l$)
13:             $L_i$.erase($l$)
14:         for $h \in H_i$ do
15:             $f \leftarrow H_i[h]$
16:             if $|f| = s$ then
17:                 $B \leftarrow B \setminus \{(i, k, L_i)\} \cup \{(i, k + 2^j, f)\}$
18:             else if $|f| \geq 2$ then
19:                 $m \leftarrow m + 1$
20:                 $L_i$.append($m$)
21:                 $B' \leftarrow B' \cup \{(m, k + 2^j, f)\}$
22:                 $A[m] \leftarrow A[f[1]]$
23:             else if $|f| = 1$ then
24:                 $L_i$.append($f$)
25:     $B \leftarrow B \cup B'$
26: for $(i, k, L_i) \in B$ do
27:     $L_i$.sort($l \mapsto T[A[l] + k]$)
28: return Output-Arrays($B, b, A$)
```

During the first phase (Algorithm 1, Lines 1-25), the suffixes are distributed into groups such that all suffixes belonging to a particular group share a common prefix. At the end of this process, we are left with a hierarchy of groups that describes the exact longest common prefixes between suffixes. The members of each group are then sorted lexicographically, which is made possible by knowing their longest common prefixes (Algorithm 1, Lines 26-27), such that a traversal of the hierarchy will yield the suffixes in lexicographic order. This is the second phase (Algorithm 1, Line 28 and Algorithm 2): a simple depth-first search is used to construct the sparse suffix array and accompanying sparse LCP array from the hierarchy.

3.1 Computing and Sorting the LCP Groups

During the first phase, the suffixes of A are organized into several LCP groups stored in set B. Each group in B is represented by a triple $(i, k, \{v_1, \ldots, v_{n_i}\})$,

[6] We stress that the pseudocode is complete in the sense that it only assumes the implementation of Lemma 1 (Line 11).

where i is the index (id) of the group, k is its associated LCP value and v_1 through v_{n_i} are its members, which are either suffixes or other groups. To distinguish between suffixes and groups, the indices 1 through b are reserved for the suffixes in A and the group numbering starts at $b+1$. At every point of the algorithm, it holds that in a group $(i, k, \{v_1, \ldots, v_{n_i}\})$, all suffixes and groups (with their respective suffixes) in $\{v_1, \ldots, v_{n_i}\}$ share a prefix of length *at least* k. At the start (base case), there exists just one group $(b+1, 0, \{1, \ldots, b\})$ (Line 3) containing all suffixes as members.

The LCP groups are then "refined" (the refinement process will be explained shortly) over the course of $\lfloor \log n \rfloor$ iterations, such that in the end each group describes the exact longest common prefix of its members rather than just a lower bound. Specifically, by the end of iteration j (where j descends from $\lfloor \log n \rfloor$ down to zero), in a group with LCP value k, two suffixes will have an actual longest common prefix of at least k and at most $k + 2^j - 1$ letters. This gap is closed once j has reached zero, at which point the refinement process is completed. The algorithm allows specifying a different starting value for j than $\lfloor \log n \rfloor$, through the parameter j_{start}. This is used in the parameterized algorithm described in Section 4.

The refinement process works as follows in iteration j. We refine every existing group; let one such group be $(i, k, \{v_1, \ldots, v_{n_i}\})$. We create a hash table (Line 8) and for every group member, with index v_i, we take the KR fingerprint as per Lemma 1 of $T[A[v_i] + k \mathinner{.\,.} A[v_i] + k + 2^j - 1]$ (Line 11).[7] If v_i denotes a group, we do the same thing using any given suffix belonging to that group; this is easily achieved by appending "witness" suffixes to A for every created group as seen in Lines 4 and 22. (Any suffix can be a witness but we choose the one with the smallest index.) All the members are grouped in the hash table based on their KR fingerprints: if two suffixes have the same KR fingerprint, they will end up in the same entry of the hash table and with high probability have the same prefix of length $k + 2^j$. To save space, entries are removed from the group as they are added to the hash table (Line 13). All entries of the hash table are then inspected (Line 14). We distinguish three cases. In case 1 (Lines 16-17), if all suffixes in a group end up having the same KR fingerprint, we update the LCP value of the old group to $k+2^j$ rather than creating a new group. In case 2 (Lines 18-22), if two or more suffixes have the same KR fingerprint, a new group is made with LCP value $k + 2^j$, containing these suffixes, and added to B. After removing the suffixes from their original group, we replace them with the index of the newly created group (Line 20). In case 3 (Lines 23-24), if a suffix is not grouped with any other suffix, we append it back to its original group.

Once the iteration with $j = 0$ ends, all LCP groups describe the exact longest common prefix of their members.[8] We now sort the members of every group lexicographically (Lines 26-27). Sorting can be done using merge sort or radix sort, because these algorithms can be performed in place using $\mathcal{O}(1)$ additional

[7] We assume that $A[v_i] + k + 2^j - 1 \leq n$; otherwise, the suffix ends at position n.

[8] This is generally not true when j_{start} was set to a value less than $\lfloor \log n \rfloor$; in this case, the LCP values are only correct if they are at most $2^{j_{\text{start}}+1} - 1$; see Section 4.

memory. Moreover, since we now know the exact LCP value for each group, two members in the same group can easily be compared in constant time: if they have a longest common prefix of length k, then the first position in which they differ is $k+1$, meaning they can be compared by only comparing their $k+1$-th letters. After this, set B contains the complete and sorted LCP groups, which are passed on to the second step of the algorithm.

3.2 Constructing the SSA and SLCP Array

The second phase (Algorithm 2) of the main algorithm involves traversing the groups created in the previous phase in order to construct the SSA and SLCP array. At this point, the members of each group are sorted lexicographically, which means that the SSA can be obtained by a simple pre-order walk along the hierarchy of the groups. For any two members, their exact longest common prefix is stored by their lowest common ancestor; that is, the group with the greatest LCP value that both suffixes fall under.

This part of the algorithm is thus a simple depth-first search of the underlying hierarchy that records all encountered suffixes in SSA in the order they appear. For every group that is visited, the LCP value of its direct "parent" is stored with it (Lines 4 and 17). Throughout, a value ℓ is tracked that takes the value of the lowest LCP value that has been seen since the last suffix was encountered (Lines 8-9); every time a suffix is appended to SSA, ℓ is appended to the SLCP array (Lines 11-12). This completes the construction.

Algorithm 2 OUTPUT-ARRAYS

Input: Set B of tuples (i, k, L_i) in ascending order by i, integer b, and array A
Output: SSA and SLCP

```
 1: SSA ← empty array
 2: SLCP ← empty array
 3: $S$ ← empty stack
 4: $S$.push($(b+1, 0)$)
 5: $\ell \leftarrow 0$
 6: while $S$ is not empty do
 7:     $(i, \ell') \leftarrow S$.pop()
 8:     if $\ell' < \ell$ then
 9:         $\ell \leftarrow \ell'$
10:     if $i \leq b$ then
11:         SSA.append($A[i]$)
12:         SLCP.append($\ell$)
13:         $\ell \leftarrow \infty$
14:     else
15:         $(i, k, L_i) \leftarrow B[i-b]$
16:         for $i' \in L_i$ in reverse order do
17:             $S$.push($(i', k)$)
18: return SSA and SLCP
```

3.3 Analysis

We prove the following result (Theorem 1) by analyzing the time (Lemma 2) and space (Lemma 3) complexity of MAIN-ALGO. (The correctness of the algorithm follows directly from [16].)

Theorem 1. *For any string $T \in \Sigma^n$, any set $T_{\mathcal{B}}$ of b suffixes of T, and any $s \in [b, n]$,* MAIN-ALGO *with j_{start} set to $\lfloor \log n \rfloor$ computes the SSA and SLCP of $T_{\mathcal{B}}$ in $\mathcal{O}(n + (bn/s)\log s)$ time using $s + 7b + o(b)$ machine words. The output is correct with high probability.*

Lemma 2. MAIN-ALGO *with j_{start} set to $\lfloor \log n \rfloor$ runs in $\mathcal{O}(n+(bn/s)\log s)$ time.*

Proof. The first phase of the algorithm consists of $\mathcal{O}(\log n)$ iterations and a sorting step. During every iteration, each existing group is considered and every member within the group is hashed. After being hashed, it is either re-added to the same group or put into a new group. The total number of groups is at most $b-1$, as the group structure represents a conceptual tree (hierarchy) with b leaves in which all internal nodes have at least two children. The number of members in each group is at most b. However, by amortization, it can be seen that every member (that is, every suffix and every group other than the "root") is processed precisely once during every iteration. Thus, we have $2b-2=\mathcal{O}(b)$ members in total.

For every group member, a KR fingerprint is computed. After the one-time pre-processing of T in $\mathcal{O}(n)$ time, the KR fingerprint of a length-k substring can be computed in $\mathcal{O}(\min\{k, n/s\})$ time (Lemma 1). In the first $\log s$ iterations, the cost is $\mathcal{O}(n/s)$, so the total cost of these iterations is $\mathcal{O}((bn/s)\log s)$. After $\log s$ iterations, the length k of the substring whose KR fingerprint is computed is $k < n/2^{\log s} = n/s$ and so the total cost of all remaining iterations is $bn/s + bn/(2s) + bn/(4s) + \cdots + b = \mathcal{O}(bn/s)$. Thus the total cost of computing all KR fingerprints is $\mathcal{O}(n + (bn/s)\log s)$.

Every group member has its KR fingerprint taken and added to a hash table supporting constant worst-case operations with high probability [4, 1]. Afterwards, all members from the hash table are re-added to the groups; for every KR fingerprint collision a new group is created with its respective members, and all other members are re-added to the original group. The number of newly created groups is at most half the number of members in the original group, as every new group has to contain at least two members. So other than the fingerprinting, all operations for a single member are performed in constant time with high probability,[9] meaning that the total time for every iteration is $\mathcal{O}(b\log n) = \mathcal{O}((bn/s)\log s)$.

In the sorting step, we have two cases: (a) $b < n/\log n$ and (b) $b \geq n/\log n$. The members in each group are sorted using in-place merge sort [27, 21] (Case (a)) or in-place radix sort [13] (Case (b)) in $\mathcal{O}(n)$ time.

Case (a): $b < n/\log n$. Sorting k members with merge sort takes $\mathcal{O}(k\log k)$ time. Recall that there are at most $b-1$ groups and that the total number of members over all groups is at most $2b-2=\mathcal{O}(b)$. If the number of members to be sorted in group i is k_i, then $k_1 + \cdots + k_{b-1} = \mathcal{O}(b)$ so the time needed to sort all groups is $\mathcal{O}(k_1\log k_1 + \cdots + k_{b-1}\log k_{b-1}) = \mathcal{O}((k_1+\cdots+k_{b-1})\log b) = \mathcal{O}(b\log b) = \mathcal{O}(n)$.

Case (b): $b \geq n/\log n$. We employ the algorithm by Franceschini et al. [13], which, given an array A of k $\mathcal{O}(\log k)$-bit integers, sorts A in place in $\mathcal{O}(k)$ time. Sorting the $2b-2$ members takes $\mathcal{O}(b)=\mathcal{O}(n)$ time, because every member can be encoded by its group id, which is a $\mathcal{O}(\log b)$-bit integer, and a letter, which is also a $\log\sigma = \mathcal{O}(\log n) = \mathcal{O}(\log b)$-bit integer, where $\sigma = |\Sigma|$.

The second phase of the algorithm is a simple stack-based DFS. Each of the $\mathcal{O}(b)$ members is pushed to and popped from the stack precisely once. The

[9] If this is not the case, we output incorrect arrays deliberately to ensure that our algorithm is Monte Carlo.

further operations applied to each member all take $\mathcal{O}(1)$ time, so this step takes $\mathcal{O}(b)$ time.

Adding all this together gives $\mathcal{O}(n) + \mathcal{O}(bn/s) \cdot \mathcal{O}(\log s) + \mathcal{O}(n) + \mathcal{O}(b) = \mathcal{O}(n + (bn/s)\log s)$ time. □

We remark that, like the algorithm by I et al. [16], Main-Algo can be amended to work in $\mathcal{O}(n)$ time, when $s = b \log b$.

Lemma 3. Main-Algo *can be implemented using $s + 7b + o(b)$ machine words, excluding the read-only string T, the array A representing the set of b suffixes, and the write-only output arrays SSA and SLCP.*

Proof. We analyze the peak space used by the algorithm neglecting the use of $\mathcal{O}(1)$ machine words:

- **KR fingerprints**: Pre-processing T to compute KR fingerprints takes s machine words by Lemma 1.
- **Array** A: Array A starts with b integers as input, but at most $b - 1$ more integers are appended to it during the algorithm to store witness suffixes for new groups, so it stores at most $b - 1$ extra integers. (Even if A is read-only we can simulate the append operation by using an extra array.)
- **Set** B: We implement set B using three integer arrays: K of size $b - 1$; C of size $b - 1$; and L of size $2b - 2$. $K[i]$ stores the LCP value for the group with id $i + b$; and $L[C[i-1]+1], \ldots, L[C[i]]$ are the group's member id's.[10] There are at most $b - 1$ groups; for every group one integer is stored in K and C as well as the group member id's in L. The total number of group members is at most $2b - 2$, since all groups except the "root" group are a member. Thus B can be implemented using $4b - 4$ integers.
- **Hash table** H_i: While processing group $i + b$, every group member is (removed from the group and) added to a hash table H_i as satellite value of the corresponding KR fingerprint key. We use a space-efficient hash table storing $c_i = C[i] - C[i - 1]$ integers (KR fingerprints) as keys: By using [4, 1], we implement H_i using $(1+\epsilon)c_i$ machine words, for any $\epsilon = \Omega(\log \log c_i / \log c_i)$. We need at most b integers to maintain the size of the satellite values per KR fingerprint because every group can have at most b members. By choosing $\epsilon = \log \log c_i / \log c_i$ we need at most $2b + o(b)$ machine words in total.

We can delete the fingerprint data structure and the hash table before moving to the sorting step. Sorting does not use any additional space because merge sort and radix sort can be implemented in-place [27, 21, 13], thus using only $\mathcal{O}(1)$ additional machine words. The first phase of the algorithm uses at most $s + 7b + o(b)$ machine words but at the end of it we have $5b + \mathcal{O}(1)$ machine words stored: array A and set B.

We now analyze the space used in the second phase (Algorithm 2); in particular, the space taken by the search stack. The stack stores at most every group and every suffix. However, the stack never simultaneously stores a member and

[10] If $i = 1$ then the group member id's are $L[1], \ldots, L[C[i]]$.

one of its ancestors, meaning the maximum size of the stack at any point is at most the maximum width of the sparse suffix tree, which is b. Every element in the stack consists of two integers, so the stack takes up at most $2b$ machine words. No other machine words need to be stored as the maximum stack size b is known in advance and so the stack is implemented using an array.

Adding this together gives at most $s + 7b + o(b)$ machine words in total. □

4 A Simple Parameterized Algorithm

Motivation. Let us start by motivating the parameterization. In real-world datasets, the b suffixes in $T_{\mathcal{B}}$ will generally not share very long prefixes. Even when they do, it is highly unlikely that all of them have this property. While Main-Algo is theoretically efficient, it would waste a lot of time with such datasets by considering large overlaps between suffixes when in reality the longest common prefixes are much shorter or when only very few suffixes share very long prefixes. Below, we show a simple method to take advantage of this, by only considering short common prefixes in the beginning and then extending them *only* for the suffixes that happen to share longer prefixes. By considering an extra parameter b' indicating the number of suffixes that share longest common prefixes longer than a certain threshold, we arrive at a time and space complexity that appears favorable for such real-world datasets.

Main Idea. We design an algorithm for constructing SSA and SLCP which is parameterized by the total number $b' \leq b$ of suffixes which have an LCP value of at least $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ with some other suffix. We show that partitioning the b suffixes into two classes (one with suffixes with LCP value strictly less than ℓ; and another with suffixes with LCP value greater than or equal to ℓ) can be done in $\mathcal{O}(n)$ time. In particular, we show that it suffices to invoke Theorem 1 twice: once (with a small change) for the b suffixes; and once (as is) for the b' suffixes; and then merge the partial results in $\mathcal{O}(b)$ time to obtain the final SSA and SLCP array.

Description and Pseudocode. The pseudocode is given as Parameterized-Algo (Algorithm 3); it is complete in the sense that it only assumes the implementation of Main-Algo. A line-by-line explanation of the algorithm follows.

Parameterized-Algo invokes the original algorithm Main-Algo twice with different arguments. In Line 1, it calls Main-Algo with the full array A as argument (and $s = b$). We set the parameter j_{start} that indicates the starting value of j (Line 5 of Algorithm 1) to $\lfloor \log \frac{n}{b} \rfloor$, meaning that j starts at a lower value than the value $\lfloor \log n \rfloor$ used in the Main-Algo and so it will take less time to complete. The result of this is that SSA will only be sorted up to $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ positions. This means that for every consecutive pair of suffixes in SSA, if their LCP value is less than ℓ, they will already be sorted correctly, whereas the other suffixes, with associated LCP values of ℓ, will need to be further sorted in the second phase (Lines 8 to 13) of Parameterized-Algo.

Algorithm 3 Parameterized-Algo

Input: string $T \in \Sigma^n$, integer b, and array A of b integers

Output: SSA and SLCP

```
1:  $\mathsf{SSA}, \mathsf{SLCP} \leftarrow$ Main-Algo$(T, A, b, j_{\text{start}} = \lfloor \log \frac{n}{b} \rfloor)$
2:  $\ell \leftarrow 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$
3:  $P, A' \leftarrow$ empty arrays
4:  for $i = 1, \dots, b$ do
5:      if $\mathsf{SLCP}[i] = \ell \vee (i < b \wedge \mathsf{SLCP}[i+1] = \ell)$ then
6:          $P$.append$(i)$
7:          $A'$.append$(\mathsf{SSA}[i])$
8:  if $|A'| > 0$ then
9:      $\mathsf{SSA}', \mathsf{SLCP}' \leftarrow$ Main-Algo$(T, A', |A'|)$
10:     for $i = 1, \dots, |A'|$ do
11:         $\mathsf{SSA}[P[i]] \leftarrow \mathsf{SSA}'[i]$
12:         if $\mathsf{SLCP}[P[i]] = \ell$ then
13:             $\mathsf{SLCP}[P[i]] \leftarrow \mathsf{SLCP}'[i]$
14: return $\mathsf{SSA}$ and $\mathsf{SLCP}$
```

What remains is to identify the suffixes that need to be further sorted, sort these suffixes separately from the others, and re-insert them into the output arrays along with the corrected LCP values. We use two arrays A' and P for this purpose: A' contains the suffixes; and P tracks the positions in SSA that these suffixes are taken from, to ensure that they will later be re-inserted at the correct positions. In Line 5, we ensure that the right suffixes are tracked in these arrays, namely those that have an LCP value of ℓ with their predecessor or successor suffix. If any such suffixes are found, we invoke Main-Algo again (Line 9), but with just these suffixes (those in array A') as input, and with the default value of $j_{\text{start}} = \lfloor \log n \rfloor$. This means that the suffixes of A' will now be fully sorted rather than being sorted up to ℓ positions. Then, in Lines 10 and 11, we insert these re-sorted suffixes at the same positions that they were taken from before, but in the corrected order. In Lines 12 and 13, we also copy the associated LCP values, but only at the positions in-between two re-sorted suffixes, as all other LCP values were already correct.

We next state and prove Theorem 2.

Theorem 2. *For any string $T \in \Sigma^n$ and any set $T_{\mathcal{B}}$ of b suffixes of T,* Parameterized-Algo *computes the SSA and SLCP of $T_{\mathcal{B}}$ in $\mathcal{O}(n + (b'n/b) \log b)$ time using $8b + 4b' + o(b)$ machine words, where b' is the total number of i such that $\mathsf{SSA}[i] \in \mathcal{B}$ and $\mathsf{SLCP}[i] \geq \ell$ or $\mathsf{SLCP}[i{+}1] \geq \ell$, with $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$. The output is correct with high probability. When $b' = \mathcal{O}(b / \log b)$,* Parameterized-Algo *runs in $\mathcal{O}(n)$ time using $8b + o(b)$ machine words.*

Time Complexity. The first phase of the algorithm (Line 1) runs in $\mathcal{O}(\log \frac{n}{b})$ iterations. The longest prefixes whose KR fingerprints are computed have length $\mathcal{O}(\frac{n}{b})$, and there are $\mathcal{O}(b)$ KR fingerprints computed in each iteration. This means that computing the KR fingerprints during the first phase takes $\mathcal{O}(b) \cdot (\mathcal{O}(\frac{n}{b}) + \mathcal{O}(\frac{n}{2b}) + \mathcal{O}(\frac{n}{4b}) + \dots) = \mathcal{O}(n)$ time. Hashing the fingerprints takes $\mathcal{O}(b \log \frac{n}{b}) = \mathcal{O}(n)$ worst-case time in total with high probability. (Grouping the fingerprints via distribute-and-collect, like the algorithm by I et al. [16], would incur a multiplicative factor of $\log_s n$.) Sorting takes $\mathcal{O}(n)$ time (see Lemma 2). Therefore the entire first phase runs in $\mathcal{O}(n)$ time. The second phase (Lines 8 to 13) computes KR fingerprints of longer prefixes as well and otherwise runs the same as Main-Algo, with the exception that only b' suffixes are now sorted. By Lemma 2, for $s = b$, this takes $\mathcal{O}(n + (b'n/b)\log b)$ time. All other operations run in single loops over arrays of size b or b' with constant-time operations, and thus take $\mathcal{O}(b)$ time. Adding everything together gives $\mathcal{O}(n + (b'n/b)\log b)$ time. When $b' = \mathcal{O}(b/\log b)$, the running time becomes $\mathcal{O}(n)$.

Space Complexity. The first phase of the algorithm uses $s + 7b + o(b)$ machine words (Lemma 3). The additional arrays P, A', SSA' and SLCP' use $4b'$ machine words in total. The second invocation of Main-Algo uses $s + 7b' + o(b')$ machine words (Lemma 3). By setting $s = b$, the algorithm uses $8b + 4b' + o(b)$ machine words in total. If $b' = \mathcal{O}(b/\log b)$, the algorithm uses $8b + o(b)$ machine words.

Correctness. We prove the correctness of SSA by Lemma 6 and that of SLCP by Lemma 7. To prove these lemmas, we first show the auxiliary Lemmas 4 and 5.

Lemma 4. *Let SSA_1 be the instance of SSA after the first invocation of* Main-Algo *(Line 1). The strings $T[\mathsf{SSA}_1[i]\,..\,n]$, $i \in [1, b]$, are sorted up to their prefix of length $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$.*

Proof. In Main-Algo, all LCP values can be increased by powers of two in each iteration. With the starting value $j_{\text{start}} = \lfloor \log \frac{n}{b} \rfloor$, this adds up to a maximum LCP value of ℓ in any group. At any point during Main-Algo, two suffixes that are in the same group with LCP value k share a longest common prefix of length at least k. Thus, this invocation of Main-Algo will compute the LCP values between suffixes correctly if they are at most ℓ, and all other LCP values will be ℓ. The sorting step takes into account only the letter which appears after the computed (longest) common prefix, so if the LCP between any two suffixes is less than ℓ the suffixes are sorted correctly. □

Lemma 5. *Let SSA_1 be the instance of SSA after the first invocation of* Main-Algo *(Line 1), and let SSA_2 be the instance of SSA returned at the end of* Parameterized-Algo *(Line 14). For every $i \in [1, b]$, either $\mathsf{SSA}_1[i] = \mathsf{SSA}_2[i]$ and $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ have a longest common prefix of length $n - \mathsf{SSA}_1[i] + 1$, or $\mathsf{SSA}_1[i] \neq \mathsf{SSA}_2[i]$ and $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ have a longest common prefix of length at least ℓ.*

Proof. If $\mathsf{SSA}_1[i] = \mathsf{SSA}_2[i]$, this is trivial, so we only concern ourselves with the case $\mathsf{SSA}_1[i] \neq \mathsf{SSA}_2[i]$. In this case, the value was overwritten in Line 11, meaning that the suffix $\mathsf{SSA}_1[i]$ was stored in A' in Line 7 to be re-sorted in the second invocation of Main-Algo. The same must hold for $\mathsf{SSA}_2[i]$.

Consider the array A' as it is built in Lines 4-7. By Lemma 4, the suffixes of SSA_1 are sorted up to their length-ℓ prefix; since the entries of A' appear in the same order as they appear in SSA_1, this must also be the case for A'. Because the suffixes of A' are already sorted correctly up to their length-ℓ prefix, it must be that for every position $j \in [1, b']$, $A'[j]$ and $\mathsf{SSA}'[j]$ have the same length-ℓ prefix. Now note that if $\mathsf{SSA}_1[i]$ appears in position j in A', then $\mathsf{SSA}_2[i]$ will take the value from $\mathsf{SSA}'[j]$. Since $A'[j]$ and $\mathsf{SSA}'[j]$ have a length-ℓ common prefix, $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ must as well. □

Lemma 6. *The instance SSA_2 of SSA returned at the end of* Parameterized-Algo *(Line 14), contains the suffixes of A sorted lexicographically.*

Proof. We prove this by showing that for any two consecutive positions i and $i+1$, $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ appear in the right order. Let SSA_1 be the instance of SSA after the first invocation of Main-Algo.

We already know that SSA_1 is sorted correctly up to ℓ positions. This means that for any i, if the longest common prefix of $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ is shorter than ℓ, they already appear in the correct order in this array. If neither suffix is overwritten after the second phase, this is also trivially the case for them in SSA_2. Now suppose that exactly one of the two (wlog $\mathsf{SSA}_1[i+1]$) is replaced by some other suffix s while the other remains the same. Let k be the LCP of $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$. By Lemma 5, $\mathsf{SSA}_1[i+1]$ and s have a longest common prefix of length at least ℓ. This is longer than k, which is strictly less than ℓ. This means that the $(k+1)$-th letter of s is the same as that of $\mathsf{SSA}_1[i+1]$, which is the first position in which it differs from $\mathsf{SSA}_1[i]$. Thus $\mathsf{SSA}_2[i] = \mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i+1] = s$ are sorted correctly relative to one another.

The remaining case is when $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ have a longest common prefix of length ℓ or longer. In this case, both suffixes are added to A' to be re-sorted in the second invocation, and both $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ may take the value of another suffix. The second invocation of the main algorithm sorts all suffixes in A' completely, returning SSA'. The suffixes in SSA' are then re-inserted into SSA_2, in which they will appear in the same order as they did in SSA'. Therefore, no matter which suffixes end up at $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$, they also appeared consecutively in SSA' and therefore must be sorted correctly. □

Lemma 7. *For any two consecutive positions i and $i+1$, $\mathsf{SLCP}[i+1]$, as returned by Algorithm 3, gives the length of the longest common prefix of $\mathsf{SSA}[i]$ and $\mathsf{SSA}[i+1]$.*

Proof. Let SSA_1 and SLCP_1 be the arrays returned by the first invocation of Main-Algo, and SSA_2 and SLCP_2 the arrays produced at the end. By Lemma 4, if $\mathsf{SLCP}_1[i+1] < \ell$, this value is correct. Therefore, the only values that need to be overwritten for SLCP_2 are when $\mathsf{SLCP}_1[i+1] = \ell$. The check at Line 12 ensures this. Of course, when $\mathsf{SLCP}_1[i+1] = \ell$, then both $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ are added to A' in order to be re-sorted in the second invocation. The values at $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ are then replaced by two suffixes that appear consecutively in SSA', say $\mathsf{SSA}'[j]$ and $\mathsf{SSA}'[j+1]$. By the correctness of Main-Algo, the LCP value of these two suffixes is given by $\mathsf{SLCP}'[j+1]$, which is the value that $\mathsf{SLCP}_2[j+1]$ takes. □

Random Strings. Finally, we show that Parameterized-Algo can be trivially amended to work in $\mathcal{O}(n)$ time for any string chosen uniformly at random from Σ^n. In particular, we show the following result.

Theorem 3. *For any string T chosen uniformly at random from Σ^n and any set $T_{\mathcal{B}}$ of b suffixes of T, SSA and SLCP of $T_{\mathcal{B}}$ can be computed in $\mathcal{O}(n)$ time using $\mathcal{O}(b)$ space. The output is correct with high probability.*

Proof. We assume $|\Sigma| \geq 2$, otherwise the problem has a trivial solution. Bollobás and Letzter [8, Theorem 4] showed that the maximum length of an LCE on T is at most $2\log_{|\Sigma|} n + \log_{|\Sigma|}\log_{|\Sigma|} n$ with high probability. We bound this from above by $3\log n$ and amend Parameterized-Algo as follows:

Case (a): $b\log n < n$. We invoke Main-Algo by setting j_{start} to the smallest integer such that $2^{j_{\text{start}}} \geq 2\lfloor\log n\rfloor$, which gives $\ell = 2\lfloor\log n\rfloor \cdot 2 - 1 = 4\lfloor\log n\rfloor - 1$. After the $\mathcal{O}(n)$-time preprocessing of Lemma 1, computing the KR fingerprints takes $\mathcal{O}(b)\cdot 4(\mathcal{O}(\frac{\log n}{1})+\mathcal{O}(\frac{\log n}{2})+\mathcal{O}(\frac{\log n}{4})+\ldots) = \mathcal{O}(b\log n)$ time. Hashing the fingerprints takes $\mathcal{O}(b)$ time per iteration with high probability, and so $\mathcal{O}(b\log n)$ total time. Merge sort takes $\mathcal{O}(b\log b)$ time. Since $\ell > 3\log n$, all suffixes of $T_{\mathcal{B}}$ will be fully sorted from the first invocation of Main-Algo. If $b' = \mathcal{O}(b/\log b)$ suffixes are still unsorted after the first invocation, these will be fully sorted in the second invocation of Main-Algo in $\mathcal{O}(n)$ time (Theorem 2). If $b' = \omega(b/\log b)$, we output incorrect arrays. The total time complexity is thus $\mathcal{O}(n + b\log n) = \mathcal{O}(n)$. The total space used is the space used by Main-Algo, which is $\mathcal{O}(b)$.

Case (b): $b\log n \geq n$. Assume that we have $\mathcal{O}(s)$ space to sort the b suffixes; we can do it efficiently using radix sort because it suffices to sort all prefixes of them of length $\mathcal{O}(\log_\sigma n)$ by the Bollobas and Letzter's result, where $\sigma = |\Sigma|$ (otherwise, we output incorrect arrays). The b prefixes are each of length at most $c\log_\sigma n$, for some $c = \mathcal{O}(1)$; so radix sort takes $\mathcal{O}((b+s)(c \cdot \log n/\log\sigma)\cdot(\log\sigma/\log s))$ time, because we have at most $(c\log n/\log\sigma)$ letters in every prefix, and each time we sort b letters, one from each prefix, we use $(\log\sigma/\log s)$ rounds of counting sort. Conveniently, the $\log\sigma$ terms cancel out. Then, because we set $s = b$, and by the fact that we are in the case $b \geq n/\log n$, we have that $\log n/\log s = \mathcal{O}(1)$. The total time complexity is thus $\mathcal{O}(b+s) = \mathcal{O}(b)$. The total space used is $\mathcal{O}(s) = \mathcal{O}(b)$. By comparing adjacent suffixes we compute the SLCP array within the same complexities. □

References

1. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: FOCS. pp. 787–796 (2010)
2. Ayad, L.A.K., Loukides, G., Pissis, S.P.: Text indexing for long patterns: Anchors are all you need. Proc. VLDB Endow. **16**(9), 2117–2131 (2023)
3. Ben-Nun, S., Golan, S., Kociumaka, T., Kraus, M.: Time-space tradeoffs for finding a long common substring. In: CPM. LIPIcs, vol. 161, pp. 5:1–5:14 (2020)
4. Bender, M.A., Conway, A., Farach-Colton, M., Kuszmaul, W., Tagliavini, G.: Iceberg hashing: Optimizing many hash-table criteria at once. J. ACM **70**(6) (2023)
5. Bernardini, G., Fici, G., Gawrychowski, P., Pissis, S.P.: Substring complexity in sublinear space. In: ISAAC. LIPIcs, vol. 283, pp. 12:1–12:19 (2023)
6. Bille, P., Fischer, J., Gørtz, I.L., Kopelowitz, T., Sach, B., Vildhøj, H.W.: Sparse text indexing in small space. ACM Trans. Algorithms **12**(3), 39:1–39:19 (2016)
7. Birenzwige, O., Golan, S., Porat, E.: Locally consistent parsing for text indexing in small space. In: SODA. pp. 607–626 (2020)

8. Bollobás, B., Letzter, S.: Longest common extension. Eur. J. Comb. **68**, 242–248 (2018)
9. Chan, T.M., Munro, J.I., Raman, V.: Selection and sorting in the "restore" model. ACM Trans. Algorithms **14**(2), 11:1–11:18 (2018)
10. Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. ACM Trans. Algorithms **17**(1), 8:1–8:39 (2021)
11. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable (extended abstract). In: ICALP. Lecture Notes in Computer Science, vol. 623, pp. 235–246. Springer (1992)
12. Fischer, J., I, T., Köppl, D.: Deterministic sparse suffix sorting in the restore model. ACM Trans. Algorithms **16**(4), 50:1–50:53 (2020)
13. Franceschini, G., Muthukrishnan, S., Puatracscu, M.: Radix sorting with no extra space. In: ESA. Lecture Notes in Computer Science, vol. 4698, pp. 194–205. Springer (2007)
14. Gawrychowski, P., Kociumaka, T.: Sparse suffix tree construction in optimal time and space. In: SODA. pp. 425–439 (2017)
15. Grabowski, S., Raniszewski, M.: Sampled suffix array with minimizers. Softw. Pract. Exp. **47**(11), 1755–1771 (2017)
16. I, T., Kärkkäinen, J., Kempa, D.: Faster sparse suffix sorting. In: STACS. LIPIcs, vol. 25, pp. 386–396 (2014)
17. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
18. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: COCOON. Lecture Notes in Computer Science, vol. 1090, pp. 219–230 (1996)
19. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
20. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: CPM. Lecture Notes in Computer Science, vol. 2089, pp. 181–192 (2001)
21. Katajainen, J., Pasanen, T., Teuhola, J.: Practical in-place mergesort. Nord. J. Comput. **3**(1), 27–40 (1996)
22. Loukides, G., Pissis, S.P.: Bidirectional string anchors: A new string sampling mechanism. In: ESA. LIPIcs, vol. 204, pp. 64:1–64:21 (2021)
23. Loukides, G., Pissis, S.P., Sweering, M.: Bidirectional string anchors for improved text indexing and top-K similarity search. IEEE Trans. Knowl. Data Eng. **35**(11), 11093–11111 (2023)
24. Navarro, G., Prezza, N.: Universal compressed text indexing. Theor. Comput. Sci. **762**, 41–50 (2019)
25. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
26. Prezza, N.: Optimal substring equality queries with applications to sparse text indexing. ACM Trans. Algorithms **17**(1), 7:1–7:23 (2021)
27. Salowe, J.S., Steiger, W.L.: Simplified stable merging tasks. J. Algorithms **8**(4), 557–571 (1987)