

Computing Smallest Suffixient Arrays in Sublinear Time

Hiroto Fujimaru¹, Gonzalo Navarro^{2,3}, Francisco Olivares^{2,3}, Jakub Radoszewski⁴, Giuseppe Romana⁵, and Cristian Urbina⁴

¹ Department of Information Science and Technology, Kyushu University, Japan
fujimaru.hiroto.134@s.kyushu-u.ac.jp

² Department of Computer Science, University of Chile, Chile
{gnavarro, folivares}@uchile.cl

³ Center for Biotechnology and Bioengineering (CeBiB), Chile

⁴ Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland
{j.radoszewski, c.urbina-gallegos}@uw.edu.pl

⁵ Faculty of Mathematics, University of Palermo, Italy
giuseppe.romana01@unipa.it

Abstract. A suffixient array is a novel data structure that, when combined with an index providing direct access on a text T , allows us to answer a variety of pattern matching queries. In this work, we show how to compute a smallest suffixient array for $T[1..n]$ in $O(\frac{n \log \sigma}{\sqrt{\log n}} + \min(r, \bar{r}) \log^\epsilon n)$ time for any $\epsilon > 0$, where σ is the alphabet size of T and r and $\bar{r}$ are the numbers of equal-letter runs of the Burrows-Wheeler transforms of T and its reverse $\bar{T}$, respectively. This time complexity becomes sublinear when σ is small enough and $\min(r, \bar{r}) = o(\frac{n}{\log^\epsilon n})$, yielding an asymptotic improvement over state-of-the-art algorithms. We also present a series of connected algorithmic results.

Keywords: Suffixient arrays · Sublinear-time algorithms · Burrows-Wheeler transform

1 Introduction

One of the most relevant challenges in modern data compression is to represent in compressed space the huge and highly repetitive text collections that naturally arise in fields like Bioinformatics [22,23,31], so as to query them directly in compressed form. To do so, several measures of repetitiveness have been proposed to evaluate the compression effectiveness of those representations [25,26]. Among others, *suffixient arrays* [11,7] have received much attention in recent years. A suffixient array is a data structure that, once built on a text $T[1..n]$, and provided with a representation of T that provides efficient random access to it, can efficiently find one occurrence of P in T , and more generally, of each maximal substring of P that occurs in T (called “maximal exact matches”, or “MEMs”). This functionality is inferior to that of suffix arrays, but in exchange suffixient arrays can be much smaller on repetitive text collections, and they were

shown to be faster in practice than other well-known repetitiveness-aware compressed suffix arrays, such as the r -index [14].

More concretely, the size of a suffixient array is linear in the size of a *sufficient set* of T , a subset of the positions of T such that any right-extension (i.e., one-character extension of a right-maximal substring) appears aligned at its rightmost position with some suffixient set position [11]. In a way, a suffixient set captures all the different substrings appearing in the text. It is natural then to aim for suffixient sets of minimum cardinality [8], which has been called χ and studied as a repetitiveness measure in its own right [7,27,12,30].

Besides the study of χ , a relevant challenge is how to efficiently find smallest suffixient sets and compute their suffixient arrays (which are the lexicographic order of the reversed prefixes that end at the positions of the suffixient sets). Cenzato et al. showed how to find smallest suffixient sets in linear time and compressed space [8], and how to test if a given set is suffixient [9]. Other constructions are linear-time online [12], or near-real-time [21]. The smallest suffixient arrays can also be constructed in linear time [5,7].

In this paper we present the first *sublinear-time* algorithm to build smallest suffixient arrays under the RAM model of computation (where computer words of $\Omega(\log n)$ bits are manipulated in constant time). We assume that the input text, which is over an alphabet $[0..\sigma)$, comes packed into $O((n \log \sigma)/\log n)$ words, and build a smallest suffixient array in time $O(n \log \sigma / \sqrt{\log n} + r \log^\epsilon n)$ for any $\epsilon > 0$, where r is the number of runs in the Burrows-Wheeler Transform (BWT) [6] of T . Measure $r \leq n$ is also sensitive to repetitiveness; in particular it holds that $\chi \leq 2r$ [27]. More precisely, our contributions are as follows:

1. We give a new linear-time algorithm for computing a smallest suffixient array starting from the BWT, the suffix array, and the longest common prefix array of T . This offers an alternative to the algorithms of Cenzato et al. [7], which require those structures built on the reversed text instead. Our algorithm uses less working space than Fujimaru et al.'s [12] online algorithm for constructing suffixient sets. Further, it is very intuitive and easy to implement. As a first application, we use our algorithm to build suffixient arrays in compressed working space $O((r + \bar{r}) \log n)$, and time $O(n(\log(r + \bar{r}) + \log \log n))$, if the BWT of the reversed text has $\bar{r}$ runs.
2. We optimize our algorithm to work in sublinear time under the packed setting, using recent data structures [18,20] for sublinear time computation of the BWT, longest common extensions, and compressed suffix trees. We then compute a smallest suffixient array for T in $O(n \log \sigma / \sqrt{\log n} + r \log^\epsilon n)$ time, which is sublinear when $\log \sigma = o(\sqrt{\log n})$ and $r = o(n / \log^\epsilon n)$ for some $\epsilon > 0$. The working space is $O(n \log \sigma / \log n + r)$.
3. We show that, when $\log \sigma = o(\sqrt{\log n})$, this result yields an $o(n)$ -time algorithm to build a suffixient array whose size is an $O(\log^{1+\epsilon} n)$ -approximation to χ . Further, size $O(\chi)$ is not reached only when $\chi = \Omega(n / \log^\epsilon n)$, that is, when the text is not so repetitive and thus it is not very interesting to have a suffixient array instead of a plain suffix array. By weakening the ap-

proximation factor to $O(\log^{3/2+\epsilon} n)$, we reach the base construction time $O(n \log \sigma / \sqrt{\log n})$.

4. We adapt Algorithm 6 of Cenzato et al. [7] to the packed setting, achieving the same sublinear time and working space on the reverse text.
5. Combining both results, we achieve $O(n \log \sigma / \sqrt{\log n} + \min(r, \bar{r}) \log^\epsilon n)$ -time computation of a smallest suffixient array for T , within working space $O(n \log \sigma / \log n + \min(r, \bar{r}))$. Since $r = \Omega(\bar{r} \log n)$ on some string families [15], this symmetry yields strictly better complexity.
6. As a byproduct, we improve the $O(n)$ -time construction of Shibata and Bannai [30] of an $O(\chi(T))$ -sized string representation to $O(n \log \sigma / \sqrt{\log n} + r \log^\epsilon n)$ time and $O(n \log \sigma / \log n + r)$ space.

2 Terminology

We consider an alphabet $\Sigma = [0, \sigma)$ of size σ . A *string* $x[1..n]$ of length n is a finite sequence $x[1]x[2] \cdots x[n]$ of n elements in Σ . The *empty string* ε is the unique string of length 0. We let Σ^* be the set of all strings over Σ , $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, and Σ^k the set of strings of length k for $k \geq 0$. We let $x[i..j] = x[i]x[i+1] \cdots x[j]$ if $1 \leq i \leq j \leq n$; if $j < i$, we let $x[i..j] = \varepsilon$, and use curved brackets to exclude the borders, e.g. $x(i..j) = x[i+1] \cdots x[j-1]$. The concatenation of two strings $x[1..n]$ and $y[1..m]$ is the string $x \cdot y = xy = x[1] \cdots x[n]y[1] \cdots y[m]$. A *substring* (or *factor*) of x is any string y such that $x = vyz$ for some $v, z \in \Sigma^*$. A *prefix* (resp. *suffix*) of x is any substring y such that $x = yz$ (resp. $x = zy$) for some string $z \in \Sigma^*$. We denote $\mathcal{F}_x(k)$ the set of substrings of length k of x for any $k \geq 0$, and $\mathcal{F}_x$ the set of substrings of x of any possible length. The *reverse* of a string $x[1..n]$, denoted as $\bar{x}$, is the string $x[n]x[n-1] \cdots x[1]$. The *lexicographic order* between two strings is defined as follows: $x < y$ if $y = xz$ with $z \in \Sigma^+$, or $x = zax'$ and $y = zby'$ for some $z, x', y' \in \Sigma^*$ and $a, b \in \Sigma$ such that $a < b$. The *co-lexicographic order* is defined as $x <_{\text{co}} y \iff \bar{x} < \bar{y}$.

We denote a string $T[1..n]$ and call it a *text* to emphasize it is the target string to be compressed or indexed. Further, we assume $T[n] = \$$, where $\$$ is a special symbol that appears (only) at the end of T and is smaller than any other symbol in Σ . We use lowercase letters like x, y, z to denote arbitrary strings.

The *suffix array* (*SA*) of $T[1..n]$ is an array $\text{SA}[1..n]$ such that $\text{SA}[i] = j$ if $T[j..n]$ is the i -th suffix of T in lexicographic order. By $\text{ISA}[1..n]$ we denote the inverse permutation of the suffix array, that is, an array such that $\text{ISA}[i] = j \iff \text{SA}[j] = i$ for $i, j \in [1, n]$.

The *longest common prefix* of two strings x and y , denoted $\text{lcp}(x, y)$, is the longest string that is a prefix of x and y . A *longest common extension query* $\text{LCE}(i, j)$ returns the length of the longest common prefix between the suffixes $T[i..n]$ and $T[j..n]$. The *longest common prefix array* is defined as $\text{LCP}[i] = \text{LCE}(\text{SA}[i], \text{SA}[i-1])$ for $i > 1$ and $\text{LCP}[1] = 0$.

The *Burrows–Wheeler transform* (*BWT*) [6] of a text T is a permutation obtained by sorting all the suffixes of T in lexicographic order and concatenating their preceding symbols. Formally, $\text{BWT}(T) = T[\text{SA}[1] - 1] \cdots T[\text{SA}[n] - 1]$ where

we let $T[0] = T[n]$. For any string x , its *run-length encoding* is the unique sequence of pairs $(a_1, p_1), \dots, (a_t, p_t)$ such that $x = a_1^{p_1} \cdots a_t^{p_t}$ and $a_{i-1} \neq a_i$ for all $i \in [2, t]$. In particular, we use r to denote the size of the run-length encoding of $\text{BWT}(T)$, which is considered a measure of the repetitiveness of the text [25]. A position i in $\text{BWT}(T)$ is said to be a *BWT run-break* if $\text{BWT}[i-1] \neq \text{BWT}[i]$.

We use $\overline{\text{BWT}}$, $\overline{\text{SA}}$, $\overline{\text{LCP}}$, $\overline{\text{ISA}}$, $\overline{\text{LCE}}$ and $\bar{r}$, to denote the corresponding data structures and values for $\bar{T}$ instead of T .

2.1 Suffixient sets and supermaximal right extensions

The set of *right extensions* of T is $E_T = \{xa \in \mathcal{F}_T \mid \exists b \neq a \text{ such that } xb \in \mathcal{F}_T\}$. A *supermaximal right extension* is a right extension that is not a proper suffix of any other right extension. The set of supermaximal right extensions of T is $\text{SRE}(T) = \{x \in E_T \mid \forall y \in E_T, y = zx \Rightarrow y = x\}$.

Definition 1. A suffixient set $\mathcal{X} \subseteq [1, n]$ for $T[1..n]$ is a set of positions of T such that every supermaximal right extension $x \in \text{SRE}(T)$ appears as a suffix of some prefix $T[1..i]$ with $i \in \mathcal{X}$.

Definition 2. The measure $\chi(T)$ is the size of a smallest suffixient set for T .

It is known that $\chi(T) = |\text{SRE}(T)|$ [7].

Definition 3. A suffixient array A is a co-lexicographically sorted suffixient set. That is, if S is a suffixient set, then $A[j] = i$ if and only if $T[1..i]$ is the j -th prefix on the set $\{T[1..i] \mid i \in S\}$ under $<_{\text{co}}$.

Example 1. Let $T = 0100\underline{10}10\underline{9}$, where the underlined positions correspond to a smallest suffixient set $S = \{4, 7, 9\}$. The prefixes corresponding to those positions satisfy $T[1..9] <_{\text{co}} T[1..4] <_{\text{co}} T[1..7]$, hence a suffixient array is $A = [9, 4, 7]$.

2.2 Computational model

We assume a computer word of size $w = \Omega(\log n)$, where $n = |T|$. The *packed representation* of a text $T \in [0, \sigma)^n$ is a list obtained by storing $\Theta(\log_\sigma n)$ letters per machine word, thus representing T in $O(n/\log_\sigma n) = O(n \log \sigma / \log n)$ machine words. If T is given in the packed representation, we say that T is a *packed string*. We rely on the following results on packed strings, which yield sublinear times on small enough alphabets, that is, when $\log \sigma = o(\sqrt{\log n})$.

Theorem 1 ([18, Thm. 5.4]). *LCE queries in a text $T \in [0, \sigma)^n$ with $\sigma = n^{O(1)}$ can be answered in $O(1)$ time after $O(n/\log_\sigma n)$ -time preprocessing of the packed representation of T .*

Theorem 2 ([18, Thm. 6.3]). *Given a packed text $T \in [0, \sigma)^n$, the BWT of T can be constructed in $O(n \log \sigma / \sqrt{\log n})$ time and $O(n/\log_\sigma n)$ space.*

Theorem 3 ([20, Prop. 5.29 and 5.30]). *Let $\epsilon > 0$ be any constant. Given a packed text $T \in [0, \sigma)^n$, we can construct a data structure in $O(n \log \sigma / \sqrt{\log n})$ time and $O(n/\log_\sigma n)$ working space, that can compute $\text{SA}[i]$ and $\text{ISA}[i]$ for any given index i in $O(\log^\epsilon n)$ time.*

3 Computing the suffixient array in linear time

In this section, we present a linear time algorithm that computes a smallest suffixient array starting from BWT, SA, LCP, and a longest common extension data structure $\overline{\text{LCE}}$ on the reversed text. Although this algorithm is unlikely to be faster than other state-of-the-art algorithms, it is conceptually very simple, and can be sped up by using proper sublinear-space data structures. In the following sections we show how to exploit this property in order to get an asymptotic improvement over state-of-the-art algorithms on most compressible texts.

We start by introducing a set of positions of T that is suffixient (though not minimal in general), characterized by the following lemma.

Lemma 1. *Let xa be a supermaximal right extension of T , for $a \in \Sigma$. Then, there exists $i \in [1, n]$ such that $T[\text{SA}[i].. \text{SA}[i] + |x|] = xa$ and at least one of the following conditions is satisfied:*

- $\text{BWT}[i - 1] \neq \text{BWT}[i]$ and $\text{LCE}(\text{SA}[i - 1], \text{SA}[i]) = |x|$ or
- $\text{BWT}[i + 1] \neq \text{BWT}[i]$ and $\text{LCE}(\text{SA}[i + 1], \text{SA}[i]) = |x|$.

Proof. As xa is a supermaximal extension of T , there exists $b \in \Sigma$ such that $b \neq a$ and $xa, xb \in \mathcal{F}_T$. Let us consider an integer interval $U = \{i : T[\text{SA}[i].. \text{SA}[i] + |x|] = xa\}$ and let j be any index such that $T[\text{SA}[j].. \text{SA}[j] + |x|] = xb$.

Assume first that $j < k$ for all $k \in U$. Then let $i = \min U$. By the definition of U , $T[\text{SA}[i].. \text{SA}[i] + |x|] = xa$ holds. By the lexicographic order, we have $\text{LCE}(\text{SA}[i - 1], \text{SA}[i]) \geq \text{LCE}(\text{SA}[j], \text{SA}[i]) = |x|$ and $i - 1 \notin U$, so $T[\text{SA}[i - 1] + |x|] = c \neq a$ and thus $\text{LCE}(\text{SA}[i - 1], \text{SA}[i]) = |x|$. Finally, we have $\text{BWT}[i - 1] \neq \text{BWT}[i]$ as otherwise xa would be a suffix of the string $\text{BWT}[i]xa \in \mathcal{F}_T$, which is a right extension because $\text{BWT}[i - 1]xc = \text{BWT}[i]xc \in \mathcal{F}_T$.

The case $j > k$ for all $k \in U$ is symmetric; we then have $\text{LCE}(\text{SA}[i + 1], \text{SA}[i]) = |x|$ and $\text{BWT}[i + 1] \neq \text{BWT}[i]$ for $i = \max U$ (so $T[\text{SA}[i].. \text{SA}[i] + |x|] = xa$). $\square$

Definition 4. *Let $\mathcal{S}$ be the following set of substrings of T :*

$$\mathcal{S} = \{T[\text{SA}[i - 1].. \text{SA}[i - 1] + \text{LCP}[i]], T[\text{SA}[i].. \text{SA}[i] + \text{LCP}[i]] : \text{BWT}[i - 1] \neq \text{BWT}[i]\}.$$

See Fig. 1. Note that Lemma 1 implies the following observation.

Observation 1. The set $\{\text{SA}[i - 1] + \text{LCP}[i], \text{SA}[i] + \text{LCP}[i] : \text{BWT}[i - 1] \neq \text{BWT}[i]\}$ is suffixient.

Lemma 2. *A string u is a supermaximal right extension of T if and only if $u \in \mathcal{S}$ and there is no string $v \in \mathcal{S}$ such that u is a proper suffix of v .*

Proof. By the definition of $\mathcal{S}$, each element of $\mathcal{S}$ is a right extension of T .

($\Rightarrow$) If u is a proper suffix of a right extension $v \in \mathcal{S}$, then u is not supermaximal.

($\Leftarrow$) Assume $u \in \mathcal{S}$ is not a proper suffix of any string $v \in \mathcal{S}$. Then, by Lemma 1, it is not a proper suffix of any supermaximal right extension. Now

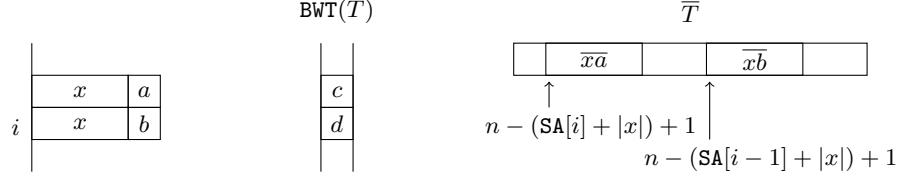


Fig. 1. Processing the BWT run-break at position i of the set $\mathcal{S}$. Substrings $T[\text{SA}[i] \dots \text{SA}[i] + |x|]$ and $T[\text{SA}[i-1] \dots \text{SA}[i-1] + |x|]$ are candidates to be supermaximal extensions of T . We add the pairs $(n - (\text{SA}[i] + |x|) + 1, |x| + 1)$ and $(n - (\text{SA}[i-1] + |x|) + 1, |x| + 1)$ to the multiset $\bar{\mathcal{S}}$, which represents the starting positions and length of the occurrences of those reversed candidates in $\bar{T}$.

assume that u is a proper suffix of a right extension v_1 that is not supermaximal. Since v_1 is not supermaximal, it is a proper suffix of a longer right extension v_2 , and so on. We continue this process until we reach a supermaximal string v that has all the previous ones—hence, u —as suffixes. $\square$

By Lemma 2, to obtain a suffixient set of the smallest cardinality, it suffices to remove the elements of $\mathcal{S}$ that correspond to proper suffixes of another element of $\mathcal{S}$. Checking this condition is equivalent to checking if an element is a proper prefix of another, when both are read backwards.

Let $\bar{\mathcal{S}}$ be the set of reversals of substrings in $\mathcal{S}$, which we treat as substrings of $\bar{T}$. We say that a substring $\bar{T}[n+1-j-\ell \dots n+1-j]$ is *reverse supermaximal* if $T[j \dots j+\ell]$ is supermaximal in T . By Lemma 1, every reverse supermaximal substring is contained in $\bar{\mathcal{S}}$. Our goal is to filter out the remaining substrings from $\bar{\mathcal{S}}$. The next corollary follows from Lemma 2.

Corollary 1. *String $X \in \bar{\mathcal{S}}$ is reverse supermaximal if and only if there is no other substring $Y \in \bar{\mathcal{S}}$ that is prefixed by X .*

This leads to the problem given next. In the problem, we remove strings being proper prefixes of other strings and leave one string from many equal copies.

Deleting prefixes among a set of substrings. We define the following auxiliary problem. Among equal substrings, we retain an arbitrary one.

DELETING PREFIXES

Input: A sequence $T[s_1 \dots e_1), \dots, T[s_t \dots e_t)$ of substrings of T

Output: List in lexicographic order all different substrings $T[s_i \dots e_i)$ such that $T[s_i \dots e_i)$ is not a proper prefix of $T[s_j \dots e_j)$ for all $j \in [1, t] \setminus \{i\}$

Example 2. If the substrings are 010, 01, 1010, 010, 1, 101, the output should contain substrings 010, 1010. In the problem, substrings are represented in $O(1)$ space as position pairs.

This problem can be solved using weighted ancestor queries on the suffix tree [2]. Below we show an arguably simpler algorithm. More importantly, in the next section we show how to make it work in sublinear time (for sufficiently small t). Notably, our algorithm does not require to sort the substrings lexicographically; it suffices to sort the corresponding suffixes of T .

We say that index i is a (proper) prefix of (or is equivalent to) index j if $T[s_i \dots e_i]$ is a (proper) prefix of $T[s_j \dots e_j]$ (or $T[s_i \dots e_i] = T[s_j \dots e_j]$, resp.).

In the solution to the problem, we store a decremental doubly-linked list of the indices in $[1, t]$. We consider the substrings in the order of non-decreasing lengths. When an index is considered, we only check if it is a prefix of any of its neighbours in the list and then remove it from the list. Technically, the list is represented as *next/prev* pointers; see Algorithm 1.

Algorithm 1: Solution to DELETING PREFIXES

```

1 Reorder the substrings so that  $T[s_1 \dots n] \leq \dots \leq T[s_t \dots n]$ ;
2 for  $i \leftarrow 1$  to  $t - 1$  do
3    $next[i] \leftarrow i + 1$ ;
4    $prev[i + 1] \leftarrow i$ ;
5  $next[t] \leftarrow prev[1] \leftarrow \infty$ ;
6 for  $i \leftarrow 1$  to  $t$  do  $good[i] \leftarrow \text{true}$ ;
7 foreach  $(e_i - s_i, i)$  in non-decreasing order do
8   foreach  $j \in \{prev[i], next[i]\}$  do
9     if  $j \neq \infty$  and  $lcp(T[s_i \dots e_i], T[s_j \dots e_j]) = e_i - s_i$  then
10     $good[i] \leftarrow \text{false}$ ;
11     $next[prev[i]] \leftarrow next[i]$ ;
12     $prev[next[i]] \leftarrow prev[i]$ ;
13 return  $\{i \in [1, t] : good[i] = \text{true}\}$ ;
```

Lemma 3. m integers in $[0, n]$ can be sorted in $O(\sqrt{n} + m)$ time.

Proof. We treat an integer $x \in [0, n]$ as a pair $(\lfloor x/\sqrt{n} \rfloor, x \bmod \lfloor \sqrt{n} \rfloor)$. We sort the resulting m pairs using Radix Sort in $O(\sqrt{n} + m)$ time [10]. $\square$

Lemma 4. Algorithm 1 solves the DELETING PREFIXES problem in $O(\min(t + \sqrt{n}, t \log t) + tq)$ time provided that any LCE and ISA query on T is answered in $O(q)$ time.

Proof. Correctness: First we show that $good[i] = \text{false}$ at the conclusion of the algorithm if and only if for some $j \in [1, t] \setminus \{i\}$, (1) i is a proper prefix of j or (2) j is equivalent to i and $j > i$.

($\Rightarrow$) If $good[i] = \text{false}$, let j be the neighbour of i in the current list that caused this assignment. Naturally, we have $j \in [1, t] \setminus \{i\}$. If i is a proper prefix of j , condition (1) is satisfied. If i is equivalent to j , we have $e_i - s_i = e_j - s_j$, so $j > i$ by the order in which the list is processed. Hence, condition (2) is satisfied.

($\Leftarrow$) Assume that $i, j \in [1, t]$, with $i \neq j$, satisfy (1) i is a proper prefix of j or (2) j is equivalent to i and $j > i$. In either case, when index i is considered in the foreach-loop, index j is still present in the list. In case (1), we assume that $j > i$, as the opposite case is symmetric. Let $i_1 = i, i_2, \dots, i_{w-1}, i_w = j$ be all indices in between those two in the list at this moment of the algorithm. Each of the substrings $T[s_{i_l} \dots e_{i_l}]$ for $l \in [1, w]$ has length at least $e_i - s_i$, so each of them has $T[s_i \dots e_i]$ as a prefix, as otherwise $T[s_1 \dots n] \leq \dots \leq T[s_t \dots n]$ would not be satisfied. Hence, $\text{good}[i] = \mathbf{false}$ because of $i_2 = \text{next}[i]$.

Now we show that substrings are reported in lexicographic order. Assume that consecutive indices i, i' such that $i < i'$ are present in the output of the algorithm. As both indices are in the output, none of the strings $u := T[s_i \dots e_i]$, $v := T[s_{i'} \dots e_{i'}]$ is a prefix of the other (in particular, $u \neq v$). Hence, there is a position $q \in [1, \min(|u|, |v|)]$ such that $u[1 \dots q] = v[1 \dots q]$ and $u[q] \neq v[q]$. We have $T[s_i \dots n] \leq T[s_{i'} \dots n]$, so $u[q] < v[q]$, i.e., $u < v$.

Complexity: Sorting suffixes $T[s_i \dots n]$ by the values $\text{ISA}[s_i]$ as well as values $e_i - s_i$ takes $O(\sqrt{n} + t)$ time by Lemma 3 or $O(t \log t)$ time using merge sort. In addition, the algorithm performs $O(t)$ operations and $O(t)$ LCP (and ISA) queries. Indeed, an *lcp* query on substrings of T can be answered using an LCE query on the corresponding prefixes of T and a comparison with the substrings' lengths. $\square$

Wrap-up. Let us analyze the time complexity of the algorithm. We compute the BWT, SA and ISA for T and $\overline{\text{ISA}}$ for $\overline{T}$ in $O(n)$ time [17]. We compute in $O(n)$ time data structures for answering LCE queries on T and $\overline{\text{LCE}}$ queries on $\overline{T}$ in $O(1)$ time [3]. This allows us to compute the set $\mathcal{S}$, and hence the set $\overline{\mathcal{S}}$, in $O(n)$ time. We then use Algorithm 1 on $\overline{T}$ to identify reverse supermaximal substrings in $\overline{\mathcal{S}}$. By Lemma 4, the algorithm works in $O(n)$ time.

By definition, if $\overline{T}[n+1-(j+\ell) \dots n+1-j]$ is such a substring, then $T[j \dots j+\ell]$ is a supermaximal right extension of T . By taking all indices $j + \ell - 1$ over all such pairs, we obtain a suffixient set of the smallest cardinality.

In the output to the DELETING PREFIXES problem, the substrings (of $\overline{T}$) are sorted according to lexicographic order, so they represent a co-lexicographic order of the original substrings of T . We leave one copy from each set of equal substrings to ensure that the size of the suffixient array is smallest. Thus we have computed a smallest suffixient array of T in the desired time and space complexity, recall Def. 3.

Building in compressed space. As a first application of our algorithm, consider the particularly interesting scenario of building the suffixient array within repetitive-aware space. Prezza [28] showed how to build the run-length-encoded BWT within $O(r)$ space in time $O(n \log r)$, providing access to the BWT in time $O(\log \log(n/r))$. Gagie et al. [14] extend this construction to add structures that compute SA, ISA, and LCP in time $O(\log(n/r))$. Their construction uses $O(r \log(n/r))$ space and $O(n(\log r + \log \log(n/r)))$ time.

To apply our algorithm, we build those structures for both T and $\overline{T}$. We then compute $\mathcal{S}$ and $\overline{\mathcal{S}}$ in time $O(r \log(n/r))$, whereas the prefix deletion phase takes

$O(r \log r + r \log(n/\bar{r}))$ time. The total time is then $O(r \log n) \subseteq O(n \log r)$ and the extra working space is $O(r)$. (We improve this result later, in Corollary 4.)

Theorem 4. *A suffixient array of a string $T \in [0, \sigma)^n$, where the BWT of T has r runs and that of $\bar{T}$ has $\bar{r}$ runs, can be computed in $O(n(\log r^* + \log \log(n/r^*)))$ time using $O(r^* \log(n/r^*))$ working space, where $r^* = \max(r, \bar{r})$.*

4 An $O(p(n, \sigma) + r \log^\epsilon n)$ time algorithm for computing χ

Let $p(n, \sigma) = \frac{n \log \sigma}{\sqrt{\log n}}$ and $p'(n, \sigma) = \frac{n \log \sigma}{\log n}$. In this section, we present an algorithm that, given a packed string T , computes a smallest suffixient array in $O(p(n, \sigma) + r \log^\epsilon n)$ time for any desired constant $\epsilon > 0$, and uses $O(p'(n, \sigma) + r)$ working space. We use the algorithm from the previous section but replace the specific data structures with their sublinear-time counterparts.

Theorem 5. *A smallest suffixient array of a packed string $T \in [0, \sigma)^n$ can be computed in $O(p(n, \sigma) + r \log^\epsilon n)$ time for any desired constant $\epsilon > 0$, using $O(p'(n, \sigma) + r)$ working space. As a by-product, the algorithm computes $\text{SRE}(T)$.*

Proof. We compute the (packed) BWT of T in $O(p(n, \sigma))$ time and $O(p'(n, \sigma))$ space [18]. Since the BWT is packed in $O(p'(n, \sigma))$ space, we can identify all run breaks, in each machine word separately, in $O(p'(n, \sigma) + r)$ time: We precompute in time $O(\sqrt{n} \log n)$ a table that, for every sequence of packed symbols fitting in $\lfloor \frac{1}{2} \log n \rfloor$ bits, records the run breaks inside the sequence.

We compute the packed representation of $\bar{T}$. To this end, we reverse the list of words representing T and also need to reverse the symbols within each word. To this end, we precompute in time $O(\sqrt{n} \log n)$ a table that stores the reverse of every sequence of packed symbols fitting in $\lfloor \frac{1}{2} \log n \rfloor$ bits. Now after $O(p(n, \sigma))$ time and $O(p'(n, \sigma))$ space preprocessing, we can compute $\text{SA}[i]$ and $\overline{\text{ISA}}[i]$ for any i in $O(\log^\epsilon n)$ time [20], and after $O(p'(n, \sigma))$ time preprocessing, the LCE of any two positions in T and in $\bar{T}$ can be computed in $O(1)$ time [18].

With these data structures, the set $\mathcal{S}$, and hence the set $\overline{\mathcal{S}}$, can be computed in $O(p(n, \sigma) + r \log^\epsilon n)$ time and $O(p'(n, \sigma) + r)$ space (via $2r$ SA , $\overline{\text{ISA}}$, and LCE queries on the BWT run limits). This allows creating the lists so that Algorithm 1 runs in $O(r + \sqrt{n})$ time, by Lemma 4. See Algorithm 2. $\square$

An always-sublinear-time approximation. When $\log \sigma = o(\sqrt{\log n})$, Theorem 5 yields $o(n)$ construction time if there is constant $\epsilon > 0$ such that $r = o(n / \log^\epsilon n)$. We can ensure $o(n)$ time for any r by building an approximation, for any fixed ϵ : We first build BWT in $o(n)$ time, which yields r . If $r \leq n / \log^{\epsilon/2} n$, we can just apply Theorem 5. Otherwise, we return instead the suffixient set defined by the endpoints of $\mathcal{S}$ (Def. 4), which is of size $2r$ and available from BWT, SA, and LCP, all of which are built in $o(n)$ time. This size is an $O(\log^{1+\epsilon} n)$ -approximation to the optimal χ , because $2r = O(n)$ and $\chi \geq \delta = \Omega(r / (\log \delta \log(n/\delta)))$ [19], where δ is the normalized substring complexity [29]. When $r = \Omega(n / \log^{\epsilon/2} n)$, since

Algorithm 2: Computing a smallest suffixient array in sublinear time

Data: A packed representation L_T of a text T
Result: A smallest suffixient array A for T .

- 1 $\overline{L_T} \leftarrow \text{reverse}(L_T)$;
- 2 $\overline{\text{LCE}} \leftarrow \overline{\text{LCE}}(\overline{L_T})$;
- 3 **for** each run-break position $i \in [\rho_1, \rho_{r-1}]$ **do**
- 4 $\overline{S} \leftarrow \overline{S} \cup \{(n+1 - (\text{SA}[i-1] + \text{LCP}[i]), \text{LCP}[i] + 1)\}$;
- 5 $\overline{S} \leftarrow \overline{S} \cup \{(n+1 - (\text{SA}[i] + \text{LCP}[i]), \text{LCP}[i] + 1)\}$;
- 6 $A \leftarrow \text{DeletingPrefixes}(\overline{L_T}, \overline{\text{LCE}}, \overline{S})$;
- 7 **return** A

$\delta = \Omega(r/\log^2 n)$ [19], it holds $n/\delta = O((n/r)\log^2 n) = O(\text{polylog } n)$. Thus $\chi = \Omega(r/(\log \delta \log(\text{polylog } n))) = \Omega(n/(\log^{\epsilon/2} n \log n \log \log n)) = \Omega(n/\log^{1+\epsilon} n)$.

Further, if we switch to just returning the endpoints of Def. 4 when we detect that $r > (n \log \sigma / \sqrt{\log n}) / \log^{\epsilon/2} n$, then we ensure total time $O(p(n, \sigma))$, at the price of worsening the approximation to $O(\log^{3/2+\epsilon} n)$.

Corollary 2. *A suffixient array of a string $T \in [0, \sigma]^n$, of size $O(\chi \log^{1+\epsilon} n)$ for any desired constant $\epsilon > 0$, can be computed in $o(n)$ time using $O(p'(n, \sigma))$ working space plus the output size. The suffixient array is of size $O(\chi)$ whenever $r = O(n/\log^\epsilon n)$. Further, we can compute in $O(p(n, \sigma))$ time a suffixient array of size $O(\chi \log^{3/2+\epsilon} n)$, which is of size χ when $r = O(n \log \sigma / \log^{1/2+\epsilon} n)$.*

Replacing the r factor by $\min(r, \bar{r})$. Cenzato et al. [7, Algorithm 6] shows how to construct a smallest suffixient array for T in linear time, starting from the arrays $\overline{\text{BWT}}$, $\overline{\text{SA}}$, and $\overline{\text{LCP}}$, of the reversed text. We adapt their algorithm using the data structures of Kempa and Kociumaka [18], obtaining the following result (see Appendix A).

Theorem 6. *A smallest suffixient array of a string $T \in [0, \sigma]^n$ can be computed in $O(p(n, \sigma) + \bar{r} \log^\epsilon n)$ time for any constant $\epsilon > 0$, using $O(p'(n, \sigma) + \bar{r})$ working space.*

By combining Theorem 5 and Theorem 6, and because r and $\bar{r}$ can be computed in $O(p(n, \sigma))$ time, we obtain the following.

Corollary 3. *A smallest suffixient array of a string $T \in [0, \sigma]^n$ can be computed in $O(p(n, \sigma) + \min(r, \bar{r}) \log^\epsilon n)$ time for any desired constant $\epsilon > 0$, using $O(p'(n, \sigma) + \min(r, \bar{r}))$ working space.*

Finally, repeating the same steps of Theorem 4 over this algorithm, which uses only the structures on $\overline{T}$, we have the following improved result.

Corollary 4. *A suffixient array of a string $T \in [0, \sigma]^n$, where the BWT of $\overline{T}$ has $\bar{r}$ runs, can be computed in $O(n(\log \bar{r} + \log \log(n/\bar{r})))$ time using $O(\bar{r} \log(n/\bar{r}))$ working space.*

5 Computing $O(\chi(T))$ -Sized String Representation

Shibata and Bannai [30] showed recently that a string T can be represented in $O(\chi(T))$ space. They presented (in Thm. 15) an $O(n)$ -time algorithm computing this representation (if T is over a linear-time-sortable alphabet). We show that an $O(\chi(T))$ -sized representation of T can be computed in sublinear time (for small enough σ and r).

The next lemma follows from efficient construction of wavelet trees [1].

Lemma 5. *For a packed text $T \in \Sigma^n$, we can compute for each letter $c \in \Sigma$ a position i such that $T[i] = c$ or state that no such position exists in $O(p(n, \sigma) + \sigma \log \sigma)$ time and $O(p'(n, \sigma) + \sigma)$ space.*

Proof. A $select_a(i)$ query for T consists in computing the i th leftmost occurrence of character $a \in \Sigma$ in T (or stating that such an occurrence does not exist). A wavelet tree for T allows to answer $select_a$ queries in $O(\log \sigma)$ time [16, 24]. Thus queries $select_a(1)$ for all $a \in \Sigma$ can be answered in $O(\sigma \log \sigma)$ time with the data structure. Finally, a wavelet tree can be constructed in $O(p(n, \sigma))$ time [1] and occupies $O(p'(n, \sigma))$ space. $\square$

The representation of Shibata and Bannai is a substring equation system.

Definition 5 ([30]). *An instance of a substring equation system (SES in short) for a string $T \in \Sigma^n$ is a triple $(n, \text{Eq}, \text{Ch})$. Each element of Eq is a triple (i, j, ℓ) with $i, j \in [1, n]$ and $1 \leq \ell \leq n - \max(i, j) + 1$, representing the equation $T[i..i + \ell] = T[j..j + \ell]$. Each element of Ch is a pair (k, c) with $k \in [1, n]$ and $c \in \Sigma$, representing the equation $T[k] = c$. We say that $(n, \text{Eq}, \text{Ch})$ represents T if T is the unique string in Σ^n satisfying all constraints in Eq and Ch . The size of the system is defined as $|\text{Eq}| + |\text{Ch}|$.*

We now recall how the SES for T of size $O(\chi(T))$ is constructed [30]. The set Ch is obtained by taking one arbitrary occurrence in T of each distinct character. The set Eq includes the following equations: for every two $yx a, y' x a' \in \text{SRE}(T)$, where x is the longest common suffix of yx and $y'x$, pick arbitrary occurrences $T[s..e] = yxa$ and $T[s'..e'] = y'xa'$ and create an equation $T[s + |y|..s + |yx|] = T[s' + |y'|..s' + |y'x'|]$. Shibata and Bannai show how all these equations can be expressed by just $\chi(T) - 1$ equations using a construction based on a compacted trie of substrings of T .

Theorem 7. *For a text $T \in [0, \sigma]^n$ such that $\log \sigma \leq \sqrt{\log n}$, an SES of size $O(\chi(T))$ representing T can be computed in $O(p(n, \sigma) + r \log^\epsilon n)$ time and $O(p'(n, \sigma) + r)$ space.*

Proof. Since $\sigma \leq 2^{\sqrt{\log n}}$, the set Ch can be computed via Lemma 5 in $O(p(n, \sigma))$ time and $O(p'(n, \sigma))$ space. For Eq , we present a construction that avoids building the compacted trie (and explicitly sorting the trimmed reversals of super-maximal right extensions).

Algorithm 3: Construction of Eq

```

1 Reorder substrings  $\overline{T}[s_i \dots e_i]$  so that  $\overline{T}[s_1 \dots n] \leq \dots \leq \overline{T}[s_t \dots n]$ ;
2 Eq  $\leftarrow \emptyset$ ;
3 for  $i := 1$  to  $t - 1$  do
4    $\ell \leftarrow \overline{\text{LCE}}(s_i, s_{i+1})$ ;
5   if  $\ell > 0$  then
6     Eq  $\leftarrow \text{Eq} \cup \{(n + 2 - s_i - \ell, n + 2 - s_{i+1} - \ell, \ell)\}$ ;
7 return Eq;

```

We compute $\text{SRE}(T)$ using Theorem 5. For each $T[i \dots j] \in \text{SRE}(T)$, we construct a substring $\overline{T}[n + 2 - j \dots n + 1 - i] = \overline{(T[i \dots j])}$ (note that the last symbol of the supermaximal right extension is removed). Let $\overline{T}[s_1 \dots e_1], \dots, \overline{T}[s_t \dots e_t]$ be the resulting substrings, where $t = \chi(T)$. Then the set **Eq** is constructed using Algorithm 3 (which resembles Algorithm 1).

Algorithm 3 works in $O(t + \sqrt{n})$ time—we use Lemma 3 for sorting—after a preprocessing that computes $\overline{\text{ISA}}$ [20] and the data structure for answering $\overline{\text{LCE}}$ queries [18]. The preprocessing itself takes $O(p(n, \sigma))$ time and $O(p'(n, \sigma))$ space. The returned set satisfies $|\text{Eq}| = \chi(T) - 1$. We just need to argue that it expresses all the desired equations. The triple inserted to **Eq** corresponds to equation

$$T[n + 1 - (s_i + \ell - 1) \dots n + 1 - s_i] = T[n + 1 - (s_{i+1} + \ell - 1) \dots n + 1 - s_{i+1}],$$

i.e., $\overline{T}[s_i \dots s_i + \ell] = \overline{T}[s_{i+1} \dots s_{i+1} + \ell]$ where $\ell = \text{lcp}(\overline{T}[s_i \dots n], \overline{T}[s_{i+1} \dots n])$. Thus every triple inserted into **Eq** implies an equation that holds in $\overline{T}$ (hence, in T). Let us consider indices j, k such that $1 \leq j < k \leq t$ and denote $d = \text{lcp}(\overline{T}[s_j \dots e_j], \overline{T}[s_k \dots e_k])$. We need to argue that the equation

$$(*) \quad \overline{T}[s_j \dots s_j + d] = \overline{T}[s_k \dots s_k + d]$$

is implied by the equations in **Eq**. Because the suffixes $\overline{T}[s_1 \dots n], \dots, \overline{T}[s_t \dots n]$ are ordered lexicographically, $\overline{T}[s_i \dots s_i + d] = \overline{T}[s_j \dots s_j + d]$ holds for all $i \in [j, k]$. This implies that $\overline{\text{LCE}}(i, i + 1) \geq d$ for all $i \in [j, k - 1]$, and equation $(*)$ is implied by the equations created for $i \in [j, k - 1]$. $\square$

6 Future Work

We have shown how to compute a suffixient array for $T[1 \dots n]$ in $O(n \log \sigma / \sqrt{\log n} + \min(r, \bar{r}) \log^\epsilon n)$ time for any $\epsilon > 0$. While further reducing the first term hits conditional lower bounds as long as we build on top of the BWT [18], the second term comes from computing $2 \min(r, \bar{r})$ direct and inverse suffix array queries using structures that can be built in sublinear time [18]. It could be possible to run those queries faster, for example in batch rather than one by one.

We also plan to work further on the line of Theorem 4, by considering conversions from/to other compressed formats (see, e.g., [13]). It might be possible,

for example, to find connections with other compressed indexes that build on right-maximal strings, like CDAWGs [4].

References

1. Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.: Wavelet trees meet suffix trees. In: Indyk, P. (ed.) *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. pp. 572–591. SIAM (2015). <https://doi.org/10.1137/1.9781611973730.39>
2. Belazzougui, D., Kosolobov, D., Puglisi, S.J., Raman, R.: Weighted Ancestors in Suffix Trees Revisited. In: Gawrychowski, P., Starikovskaya, T. (eds.) *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 191, pp. 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). <https://doi.org/10.4230/LIPIcs.CPM.2021.8>
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) *LATIN 2000: Theoretical Informatics*. pp. 88–94. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/10719839_9
4. Blumer, A., Blumer, J., Haussler, D., McConnell, R.M., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. *J. ACM* **34**(3), 578–595 (1987). <https://doi.org/10.1145/28869.28873>
5. Bonizzoni, P., Gao, Y., Riccardi, B.: Constructing suffixient arrays revisited. In: Bille, P., Prezza, N. (eds.) *37th Annual Symposium on Combinatorial Pattern Matching, CPM 2026*. *LIPIcs*, vol. 369, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2026). <https://doi.org/10.4230/LIPICS.CPM.2026.30>
6. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. *Tech. Rep. 124*, Digital Equipment Corporation (1994)
7. Cenzato, D., Depuydt, L., Gagie, T., Kim, S.H., Manzini, G., Olivares, F., Prezza, N.: Suffixient arrays: a new efficient suffix array compression technique (2025), <https://arxiv.org/abs/2407.18753>
8. Cenzato, D., Olivares, F., Prezza, N.: On computing the smallest suffixient set. In: Lipták, Z., de Moura, E.S., Figueroa, K., Baeza-Yates, R. (eds.) *Proceedings of 31st International Symposium on String Processing and Information Retrieval, SPIRE 2024*. *Lecture Notes in Computer Science*, vol. 14899, pp. 73–87 (2024). https://doi.org/10.1007/978-3-031-72200-4_6
9. Cenzato, D., Olivares, F., Prezza, N.: Testing suffixient sets (2025), <https://arxiv.org/abs/2506.08225>
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 3rd Edition. MIT Press (2009), <http://mitpress.mit.edu/books/introduction-algorithms>
11. Depuydt, L., Gagie, T., Langmead, B., Manzini, G., Prezza, N.: Suffixient sets. *CoRR* **abs/2312.01359** (2023). <https://doi.org/10.48550/ARXIV.2312.01359>
12. Fujimaru, H., Navarro, G., Romana, G., Urbina, C.: Smallest suffixient sets: Effectiveness, resilience, and calculation (2025), <https://arxiv.org/abs/2506.05638>
13. Gagie, T., Goga, A., Jež, A., Navarro, G.: Space-efficient conversions from SLPs. In: Soto, J.A., Wiese, A. (eds.) *LATIN 2024: Theoretical Informatics - 16th Latin American Symposium, Part I*. *Lecture Notes in Computer Science*, vol. 14578, pp. 146–161. Springer (2024). https://doi.org/10.1007/978-3-031-55598-5_10

14. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM* **67**(1) (jan 2020). <https://doi.org/10.1145/3375890>
15. Giuliani, S., Inenaga, S., Lipták, Z., Prezza, N., Sciortino, M., Toffanello, A.: Novel results on the number of runs of the Burrows-Wheeler-transform. In: *SOFSEM. Lecture Notes in Computer Science*, vol. 12607, pp. 249–262. Springer (2021). https://doi.org/10.1007/978-3-030-67731-2_18
16. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 12–14, 2003, Baltimore, Maryland, USA. pp. 841–850. ACM/SIAM (2003), <http://dl.acm.org/citation.cfm?id=644108.644250>
17. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. *J. ACM* **53**(6), 918–936 (Nov 2006). <https://doi.org/10.1145/1217856.1217858>
18. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. p. 756–767. STOC '19, ACM (Jun 2019). <https://doi.org/10.1145/3313276.3316368>
19. Kempa, D., Kociumaka, T.: Resolution of the Burrows-Wheeler Transform conjecture. In: *Proc. 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. pp. 1014–1025 (2020). <https://doi.org/10.1109/FOCS46700.2020.00097>
20. Kempa, D., Kociumaka, T.: Breaking the $O(n)$ -barrier in the construction of compressed suffix arrays and suffix trees. In: Bansal, N., Nagarajan, V. (eds.) *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. pp. 5122–5202. SIAM (2023). <https://doi.org/10.1137/1.9781611977554.ch187>
21. Köppl, D., Kucherov, G.: Smallest suffixient set maintenance in near-real-time (2026), <https://arxiv.org/abs/2604.27548>, accepted to MFCS 2026
22. The 1000 Genomes Project Consortium: A global reference for human genetic variation. *Nature* **526**(7571), 68–74 (2015). <https://doi.org/10.1038/nature15393>
23. Medini, D., Donati, C., Tettelin, H., Massignani, V., Rappuoli, R.: The microbial pan-genome. *Current Opinion in Genetics & Development* **15**(6), 589–594 (2005). <https://doi.org/10.1016/j.gde.2005.09.006>
24. Navarro, G.: Wavelet trees for all. *J. Discrete Algorithms* **25**, 2–20 (2014). <https://doi.org/10.1016/J.JDA.2013.07.004>
25. Navarro, G.: Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Comput. Surv.* **54**(2), 29:1–29:31 (2022). <https://doi.org/10.1145/3434399>
26. Navarro, G.: Indexing Highly Repetitive String Collections, Part II: Compressed Indexes. *ACM Comput. Surv.* **54**(2), 26:1–26:31 (2022). <https://doi.org/10.1145/3432999>
27. Navarro, G., Romana, G., Urbina, C.: Smallest suffixient sets as a repetitiveness measure. In: Badkobeh, G., Radoszewski, J., Tonello, N., Baeza-Yates, R. (eds.) *Proc. 32nd International Symposium on String Processing and Information Retrieval (SPIRE 2025)*. *Lecture Notes in Computer Science*, vol. 16073, pp. 217–232. Springer (2025). https://doi.org/10.1007/978-3-032-05228-5_18
28. Prezza, N.: Compressed Computation for Text Indexing. Ph.D. thesis, University of Udine (2016)
29. Raskhodnikova, S., Ron, D., Rubinfeld, R., Smith, A.D.: Sublinear algorithms for approximating string compressibility. *Algorithmica* **65**(3), 685–709 (2013). <https://doi.org/10.1007/S00453-012-9618-6>
30. Shibata, H., Bannai, H.: String representation in suffixient set size space (2026), <https://arxiv.org/abs/2604.04377>

31. Wang, T., Antonacci-Fulton, L., Howe, K., Lawson, H.A., Lucas, J.K., Phillippy, A.M., Popejoy, A.B., Asri, M., Carson, C., Chaisson, M.J., et al.: The human pangenome project: a global resource to map genomic diversity. *Nature* **604**(7906), 437–446 (2022). <https://doi.org/10.1038/s41586-022-04601-8>

A Sublinear time adaptation of Cenzato et al.’s algorithm

Cenzato et al. [7, Algorithm 6] show how to construct a smallest suffixient array for T in linear time, starting from the array $\overline{\text{BWT}}$, the suffix array $\overline{\text{SA}}$, and the LCP array $\overline{\text{LCP}}$ of the reversed text. Additionally, it uses an array R to keep information about the last position that is candidate to be a super-maximal extension for each $c \in \Sigma$. More precisely, in $R[c].len$ is stored the $\overline{\text{LCP}}$ value for the last candidate c -run break, in $R[c].pos$ is stored its position in T , and $R[c].active$ is a flag telling if $R[c].pos$ was already added to the suffixient set (which case it is set to false) or not.

The algorithm sequentially scans $\overline{\text{BWT}}$. For a run $\overline{\text{BWT}}[i - 1 - j .. i - 1] = c^{j+1}$, it iterates from $k = i - j$ to $i - 1$ and keeps the minimum value within $\overline{\text{LCP}}[i - 1 - j .. k]$ in a variable m and it updates $\overline{\text{LF}}[c] = \overline{\text{LF}}[c] + 1$. For a run-break $\overline{\text{BWT}}[i - 1, i] = ac$, it keeps the minimum value within $\overline{\text{LCP}}[i]$ and m and stores it in m , updates $\overline{\text{LF}}[c] = \overline{\text{LF}}[c] + 1$ and calls to *eval* to evaluate if $m < R[a].len$ (such a case $R[a].pos$ is added to the suffixient set computed so far, m is stored in $R[a].len$ and $R[a].active$ is set to *false*), and it calls to *eval* to determine if $\overline{\text{LCP}}[\overline{\text{LF}}[\overline{\text{BWT}}[c]]] - 1 < R[c].len$ (such a case $R[c].pos$ is added to the suffixient set, $\overline{\text{LCP}}[\overline{\text{LF}}[\overline{\text{BWT}}[c]]] - 1$ is stored in $R[c].len$, and $R[c].active$ is set to false). Additionally, if $R[\overline{\text{BWT}}[i']].len < \overline{\text{LCP}}[i]$, then $n - \overline{\text{SA}}[i'] + 1$ is stored in $R[\overline{\text{BWT}}[i']].pos$, $\overline{\text{LCP}}[i]$ is stored in $R[\overline{\text{BWT}}[i']].len$ and $R[\overline{\text{BWT}}[i']].active$ is set to true (with $i' \in \{i - 1, i\}$). Finally, after traversing the whole $\overline{\text{BWT}}$, any active candidate stored in R is added to the suffixient set. Thus, we can group the iterations into traversing within a run to keep the minimum $\overline{\text{LCP}}$ value and update $\overline{\text{LF}}$, and performing $O(1)$ operations for each run-break.

We modify the algorithm to avoid scanning the $\overline{\text{BWT}}$ position by position and just iterate from one run-break to the next one. To achieve this, we can note that, for two consecutive run-breaks b_{i-1} and b_i , the minimum within $\overline{\text{LCP}}[b_{i-1} .. b_i]$ is exactly the length of the prefix shared by from the (b_{i-1}) -th suffix to the b_i -th suffix of T , so instead of iterating from b_{i-1} to b_i we compute $\overline{\text{LCE}}(\overline{\text{SA}}[b_{i-1}], \overline{\text{SA}}[b_i])$. In addition, if $1 < b_i - b_{i-1}$ (so, $\overline{\text{BWT}}[b_i .. b_{i-1} - 1] = c^{b_i - b_{i-1}}$), we have to count $b_i - b_{i-1} - 1$ additional occurrences of c to get $\overline{\text{LF}}[c]$ to point to $b_i - 1$ as desired (we counted one when we processed b_{i-1}), so we update $\overline{\text{LF}}[\overline{\text{BWT}}[a]] = \overline{\text{LF}}[\overline{\text{BWT}}[a]] + (b_i - b_{i-1} - 1)$. The rest of the algorithm just needs to know the positions of the run-breaks. Algorithm 5 shows how to run the algorithm with these changes using the data structures of Kempa and Kociumaka [18].

The arrays R and $\overline{\text{LF}}$ use $O(\sigma)$ space. The sublinear data structures $\overline{\text{BWT}}$, $\overline{\text{LCP}}$, and $\overline{\text{SA}}$ take $O(p'(n, \sigma))$ space in total. Thus, the space consumption of the algorithm adds up to $O(p'(n, \sigma) + \sigma) = O(p'(n, \sigma))$.

Regarding running-time, to compute $\text{occ}(T, a_i)$, we start by setting $\text{occ}(T, a_i) = 0$ for each a_i and $\text{occ}(T, \overline{\text{BWT}}[1]) = 1$. For simplicity, we assume $b_0 = 1$. Now, for

Algorithm 4: Computing eval

Data: A set of characters C , an LCP value ℓ , the candidate list R , and a suffixient set S

Result: The updated suffixient set S

```

1 foreach  $c \in C$  do
2   if  $\ell < R[c].len$  then
3     if  $R[c].active$  then
4        $S \leftarrow S \cup \{R[c].pos\};$ 
5      $R[c] \leftarrow \{\ell, 0, \text{false}\};$ 

```

Algorithm 5: Adaptation of Cenzato et al.'s linear-time algorithm [7, Algorithm 6] for computing a smallest suffixient array

Data: A text T and data structures for $\overline{\text{BWT}}$, $\overline{\text{SA}}$ and $\overline{\text{LCP}}$

Result: A smallest suffixient set S for T .

```

1  $R[1, \sigma] \leftarrow (len \leftarrow -1, pos \leftarrow 0, active \leftarrow false) \times \sigma;$ 
2  $S \leftarrow \emptyset;$ 
3  $\overline{\text{LF}}[a_1] \leftarrow 0;$ 
4 for  $i = 2, \dots, \sigma$  do  $\overline{\text{LF}}[a_i] \leftarrow \overline{\text{LF}}[a_{i-1}] + occ(T, a_{i-1});$ 
5  $\overline{\text{LF}}[\overline{\text{BWT}}[1]] \leftarrow \overline{\text{LF}}[\overline{\text{BWT}}[1]] + 1;$ 
6 for  $i = 1, \dots, \bar{r}$  do
7    $\overline{\text{LF}}[\overline{\text{BWT}}[b_i - 1]] \leftarrow \overline{\text{LF}}[\overline{\text{BWT}}[b_i - 1]] + (b_i - b_{i-1} - 1);$  /* Assuming  $b_0 = 1$  */
8    $\overline{\text{LF}}[\overline{\text{BWT}}[b_i]] \leftarrow \overline{\text{LF}}[\overline{\text{BWT}}[b_i]] + 1;$ 
9    $m \leftarrow \text{LCE}(\overline{\text{SA}}[b_{i-1}], \overline{\text{SA}}[b_i]);$  /*  $m \leftarrow \min_{j \in [b_{i-1}, b_i]} \{\overline{\text{LCP}}[j]\}$  */
10   $eval(\overline{\text{BWT}}[b_i - 1], m, R, S);$ 
11  if  $R[\overline{\text{BWT}}[b_i]].len \neq -1$  then  $eval(\overline{\text{BWT}}[b_i], \overline{\text{LCP}}[\overline{\text{LF}}[\overline{\text{BWT}}[b_i]]] - 1, R, S);$ 
12  for  $b' \in \{b_i - 1, b_i\}$  do
13    if  $R[\overline{\text{BWT}}[b']].len < \overline{\text{LCP}}[b_i]$  then
14       $R[\overline{\text{BWT}}[b']] \leftarrow (\overline{\text{LCP}}[b_i], n - \overline{\text{SA}}[b'] + 1, true);$ 
15  $eval(\Sigma, -1, R, S)$ 

```

each run-break b_i , we set $occ(T, \overline{\text{BWT}}[b_i - 1]) = occ(T, \overline{\text{BWT}}[b_i - 1]) + ((b_i - 1) - b_{i-1})$ (so we are counting $(b_i - 1) - b_{i-1}$ additional occurrences if $\overline{\text{BWT}}[b_{i-1}, b_i - 1] = c^{(b_i - 1) - b_{i-1} + 1}$, and none otherwise) and $occ(T, \overline{\text{BWT}}[b_i]) = occ(T, \overline{\text{BWT}}[b_i]) + 1$. If $b_{\bar{r}} < n$ (so, $\overline{\text{BWT}}[b_{\bar{r}}, n] = c^{n - b_{\bar{r}} + 1}$), then we set $occ(T, \overline{\text{BWT}}[b_{\bar{r}}]) = occ(T, \overline{\text{BWT}}[b_{\bar{r}}]) + (n - b_{\bar{r}})$ to count the occurrences within that last run. Since we perform $O(1)$ operations on each run-break, this step takes $O(\bar{r})$ time. The call to *eval* on Line 10 takes $O(1)$ time, because the set given as the first argument is of size 1. The last call to *eval* on Line 15 takes $O(\sigma)$ time as the set given has size $O(\sigma)$. The total running-time and space consumption are then dominated by that of building the base structures [18] on $\overline{T}$ and using them $O(\bar{r})$ times.

This algorithm computes only a smallest suffixient set. We can sort this suffixient set in $O(\sqrt{n} + \chi)$ time (using Lemma 3) by enhancing the prefixes with their co-lexicographical order, which can be retrieved when inserting elements in

S using $\overline{\text{ISA}}$ queries. This does not increase (asymptotically) the space or time of the algorithm.