Practical Linear-Time Computation of Smallest Suffixient Sets*

Francisco Olivares[1,2][0000−0001−7881−9794] and Gonzalo Navarro[1,2][0000−0002−2286−741X]

[1] CeBiB – Centre for Biotechnology and Bioengineering, Chile
[2] Department of Computer Science, University of Chile, Chile
{folivares,gnavarro}@uchile.cl

Abstract. Suffixient arrays are recent structures that have attracted attention because they offer relevant pattern matching functionality in less asymptotic space than the Run-Length BWT, the de-facto standard to index highly repetitive string collections. Various algorithms exist for building them from the suffix array data structures. We present the first construction algorithm that is (i) linear-time, (ii) one-pass over the structures, and (iii) implemented and practical. This makes the construction particularly useful on large text collections, which we demonstrate empirically by showing that it dominates the space/time tradeoff map of the implemented constructions.

Keywords: Suffix arrays, Suffixient arrays, Burrows-Wheeler Transform, Repetitive text collections

1 Introduction and Related Work

Many of the fastest-growing text collections are highly repetitive, meaning that most documents are near-copies of others: genome collections like the One Million Genome Initiative, software repositories like Software Heritage, and versioned text collections like Wikipedia are examples of this phenomenon. Effectively searching such huge collections requires using storage formats that exploit repetitiveness to reduce space while offering fast indexed searches. The Run-Length Burrows-Wheeler Transform (RLBWT) is at the root of many successful compressed indexes for this scenario [4,16,13,8]. The space of this family of indexes is dominated by r, the number of equal-letter runs in the BWT of the collection.

A very recent development [9] further pushes towards space reduction by defining a new measure $\chi \leq r$, the size of the smallest so-called *suffixient set* of the text. This is a subset of the text positions that, if sorted by the corresponding prefixes in colexicographic order (i.e., comparing right-to-left), yields the so-called *suffixient array* [5]. Provided with a method to access the compressed

* Funded by Basal Funds FB0001 and AFB240001, and Fondecyt grant 1260080, ANID, Chile.

Algorithm	linear-time	one-pass	implemented
One-pass [6]	no	yes	yes
LF [6]	yes	no	yes
FM [5]	yes	no	yes
BGR [2]	yes	yes	~~no~~ yes
Online [12]	yes	yes*	yes
Linear [11]	yes	no	no
Near-linear [15]	no	yes*	no
Ours	yes	yes	yes

Table 1. The existing algorithms to build a smallest suffixient set, and ours in this context. We implement algorithm BGR in this paper. Algorithms "Online" and "Near-linear" are one-pass over the text, not in the same sense as the others.

text, the suffixient array can efficiently search for one occurrence of a pattern in the text, or find all its maximal substrings that occur in the text. This is less than the classic pattern matching functionality of finding all the pattern occurrences, but sufficient for various applications while using less space.

Since smallest suffixient sets are meant to be used on large repetitive text collections, their efficient construction is key to their adoption. Cenzato et al. [6,5] already proposed two efficient constructions, which work on top of the suffix array (SA), the Burrows-Wheeler Transform (BWT), and the longest common prefix array (LCP) of the reversed text. Their first algorithm [6], performs one single pass over those structures (i.e., it is "one-pass") and takes time $O(n + \overline{r} \log \sigma)$, where n is the length of the text, $\overline{r}$ is the r value for the reverse text, and σ is the alphabet size. Their second algorithm [6], which we call LF, is linear-time but not one-pass, just as their third algorithm [5], which we call FM. FM is faster than LF but uses more space. Bonizzoni et al. [2] recently proposed the first algorithm that is both linear-time and one-pass, which we call BGR, but no implementation is reported. Fujimaru et al. [12] present linear-time online algorithms that work directly on the text, processing it left-to-right or right-to-left. Fujimaru et al. [11] present another linear-time algorithm, which is however not one-pass nor implemented; it uses some structures built on the text and some on its reverse. They also present the first sublinear-time construction (also not implemented). Finally, Köppl and Kucherov [15] gave a near-real time construction algorithm, without reporting an implementation.

In this context, we present the first algorithm to build smallest suffixient sets that is (i) linear-time, (ii) one-pass, and (iii) implemented. See Table 1. We empirically demonstrate its practicality by comparing it with the implemented alternatives: our algorithm outperforms previous algorithms, or matches the best ones, in both time and space. On passing, we implement BGR algorithm, which turns out to be practical but dominated by ours in both space and time.

2 Preliminaries

By $[i, j]$ we refer to all values $i, i+1, \ldots, j$, if $i \leq j$, and $[i, j] = \emptyset$, otherwise; and we define $[n] = [1, n]$. A string $T[1, n] \in \Sigma^n$ is a concatenation of n elements of an alphabet $\Sigma = \{1, \ldots, \sigma\}$. The length of T is denoted by $|T| = n$. The empty string ϵ is the only string satisfying $|\epsilon| = 0$. $T[i]$ indicates the i-th element of T, and $T[i, j] = T[i] \ldots T[j]$ is the substring of T starting at position i and ending at position j, if $1 \leq i \leq j \leq n$, and $T[i, j] = \epsilon$ otherwise. A prefix of T is a substring of the form $T[1, j]$ and a suffix is a substring of the form $T[i, n]$. T^{rev} denotes the reversal of T, that is, $T^{\text{rev}} = T[n] \ldots T[1]$.

Given two strings $T, S \in \Sigma^+$, none a prefix of the other, and a total order over Σ, the *lexicographic* order $<_{lex}$ over Σ^+ is defined as $T <_{lex} S$ if and only if $T[1, i] = S[1, i]$ and $T[i+1] < S[i+1]$, for some i. The *suffix array* of $T[1, n]$ ($\text{SA}(T)$) is a permutation of the elements of T such that $T[\text{SA}(T)[i], n] <_{lex} T[\text{SA}(T)[j], n]$ for any $1 \leq i < j \leq n$, namely $\text{SA}(T)[i]$ is the i-th suffix of T in lexicographic order. We assume T ends with a special character \$, which is smaller than any other symbol in Σ and only occurs in $T[n]$, so SA is well defined. The *longest common prefix* array of T ($\text{LCP}(T)$) is defined as $\text{LCP}(T)[i] = \ell$ if and only if $2 \leq i$ and $T[\text{SA}(T)[i-1], \text{SA}(T)[i-1]+\ell-1] = T[\text{SA}(T)[i], \text{SA}(T)[i]+\ell-1]$ and $T[\text{SA}(T)[i-1]+\ell] \neq T[\text{SA}(T)[i]+\ell]$, that is, $\text{LCP}(T)[i]$ is the length of the maximal prefix shared by the $(i-1)$-th and the i-th suffixes of T in the lexicographic order; we further define $\text{LCP}(T)[1] = 0$. The *Burrows-Wheeler Transform* array of T ($\text{BWT}(T)$) is defined as $\text{BWT}(T)[i] = T[\text{SA}(T)[i-1]]$, if $2 \leq i$, and $\text{BWT}(T)[1] = T[n]$, that is $\text{BWT}(T)[i]$ is the symbol preceding the i-th sufix of T. By SA_r, LCP_r, and BWT_r we refer to $\text{SA}(T^{\text{rev}})$, $\text{LCP}(T^{\text{rev}})$, and $\text{BWT}(T^{\text{rev}})$, respectively, where in this case we assume $T^{\text{rev}} = (T[1, n-1])^{\text{rev}}\$$.

Given an integer array $\mathcal{I}$, the *previous smaller value* array of $\mathcal{I}$ ($\text{PSV}(\mathcal{I})$) is a lenght-$|\mathcal{I}|$ array defined as $\text{PSV}(\mathcal{I})[i] = j$ if and only if $j = \max(\{k < i \mid \mathcal{I}[k] < \mathcal{I}[i]\} \cup \{0\})$, that is, j is the largest index in $[1, i-1]$ for which $\mathcal{I}$ is strictly smaller than $\mathcal{I}[i]$, if such value exists, and $j = 0$ otherwise. The *next smaller value* array, $\text{NSV}(\mathcal{I})$, is defined analogously for the smaller value with an index bigger than i (or $|\mathcal{I}| + 1$ if none exists). For brevity, by PSV and NSV we will refer to $\text{PSV}(\text{LCP}_r)$ and $\text{NSV}(\text{LCP}_r)$, respectively.

A run in BWT_r is a maximal contiguous substring that repeats a single symbol. Given $i \in [2, n]$ and $a, c \in \Sigma$, with $a \neq c$, we say that i is a c-run break if $\text{BWT}_r[i-1, i] = ac$ or $\text{BWT}_r[i-1, i] = ca$. By $box(i) = [l_i, r_i]$ we refer to the maximal interval such that $i \in [l_i, r_i]$ and $\text{LCP}_r[i] \leq \text{LCP}_r[j]$, for any $j \in [l_i, r_i]$, in other words, $box(i) = [\text{PSV}[i] + 1, \text{NSV}[i] - 1]$. By $text(i) = n - \text{SA}_r[i] + 1$ we indicate the position in T corresponding to $\text{BWT}_r[i]$. Similarly, by $bwt(j) = \text{SA}_r^{-1}[n + 1 - j]$ we denote the position in SA_r of the symbol $T[j]$.

We say the substring $T[i, j]$ ($i - 1 \leq j$) is right-maximal if $j = n$ (so $T[i, j]$ is a suffix of T) or there exist $a, b \in \Sigma$ such that $a \neq b$ and both $T[i, j] \cdot a$ and $T[i, j] \cdot b$ occur in T. Note that, by definition, ϵ is always right-maximal.

2.1 Suffixient sets

Suffixient sets were initially proposed by Depuydt et al. [9].

Definition 1 (Suffixient set [5,6,9]). *A set $S \subseteq [n]$ is* suffixient *for a string T if, for every one-character right-extension $T[i,j]$ $(j \geq i)$ of every right-maximal string $T[i,j-1]$, there exists $x \in S$ such that $T[i,j]$ is a suffix of $T[1,x]$.*

They also proposed a compressed index based on a suffixient set of a text T that can efficiently find one occurrence of any given pattern P within T, or say that P does not occur in T. The index can also find all the maximal exact matches (MEMs) of P. They show how to compute a suffixient set of size $O(\overline{r})$, where $\overline{r}$ is the number of runs in BWT_r.

Their index uses $O(|S| + g)$ words of space, where g is the size of a grammar that gives random access to the input text and S is the suffixient set they build. It follows that the smaller the suffixient set built, the smaller their index is.

Cenzato et al. [6] followed up and studied methods to build suffixient sets of smallest cardinality. They introduced three algorithms to achieve this goal. The first of them (Algorithm 1) runs in quadratic-time and is conceptually simple. The second one (Algorithm 2) runs in time $O(n+\overline{r}\log\sigma)$. The algorithm is called *one-pass* in the paper because it needs just one sequential pass over SA_r, LCP_r, and BWT_r, which enables *prefix free parsing*, a technique which allows streaming those arrays in compressed space [3]. The third algorithm (Algorithm 4) runs in $O(n)$ time and uses the same three arrays. Since it relies on the LF mapping [10], it needs random access on LCP_r, so it does not run on top of PFP. Later [5], the same authors propose another linear-time algorithm (Algorithm 7), which uses PSV/NSV arrays and, in practice, is the fastest between the four algorithms, at the cost of using more working space. In the same paper, the authors proposed an index called *suffixient array*, a sampling of the positions of the *prefix array* (the mirroring version of SA, which uses lexicographic order over the reversed suffixes) which can be binary-searched and offers fast pattern matching, if random access to the text is still available. This index outperforms many well-known ones, such as the r-index [13], the fastest index in practice offering full pattern matching (i.e., finding all the pattern occurrences, not just one).

Several other construction algorithms were proposed after the work of Cenzato et al. [6,5]. Bonizzoni et al. [2] proposed the first algorithm that is linear-time and one-pass. Fujimaru et al. [12] proposed an online algorithm that works directly on the text (so it avoids computing SA_r, LCP_r, and BWT_r), and later [11] presented another linear-time algorithm that builds some structures on the text and on its reverse, but it is not linear-time nor one-pass. They also present the first sublinear-time construction, which relies on some complex data structures proposed by Kempa and Kociumaka [14]. Finally, Köppl and Kucherov [15] gave the first near-real time construction algorithm.

Suffixient sets have also become a field of study from other perspectives. Cenzato et al.[7] showed how to determine in linear time if an arbitrary set is suffixient of smallest cardinality. The measure χ (the size of a smallest suffixient set) is also studied as a measure of repetitiveness, due to the combinatorial

properties of suffixient sets [17]. It is known that $\chi = O(\overline{r})$, and also that $\chi = O(r)$ [17], so $\chi = O(\min(\overline{r}, r))$.

3 A Simple Linear-Time Construction Algorithm

Cenzato et al. [5, Alg. 3] give a simple algorithm to compute a suffixient set of smallest cardinality. They define a set $B_{i,c}$ that contains the positions in $box(i)$ that are c-run breaks in BWT_r. Then, for a c-run break i, with $\mathrm{BWT}_r[i'] = c$ for some $i' \in \{i-1, i\}$, the algorithm relies on adding to the resulting set the position $text(i')$ if and only if it holds

$$i = \max\{j \ : \ j \in B_{i,c} \wedge \mathrm{LCP}_r[j] = \max \mathrm{LCP}_r[B_{i,c}]\}. \tag{1}$$

That algorithm spends $O(n)$ time to determine if Eq. (1) holds, and, since boxes may overlap, it spends quadratic time to run the whole algorithm. In this section we present a method to evaluate Eq. (1) in $O(1)$ time, which leads to a linear-time version of the algorithm. As a first step, we introduce the notion of *last c-candidate* positions (Definition 2). Later, we prove that *last c-candidate* positions are exactly the positions that satisfy Eq. 1 (Proposition 1). Finally, we show a modified version of the quadratic algorithm using this equivalence (Algorithm 2).

In the rest of this section, we assume there are exactly k c-run breaks in BWT_r. Let $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ be the positions of those run-breaks, and let $box(i_h) = [l_h, r_h]$, for each $h \in [k]$.

Definition 2 (*last c-candidate*). *Let i_h be the position of a c-run break. We say i_h is a c-candidate if one of the following conditions holds:*

- $h = 1$,
- $i_{h-1} < l_h$, *or*
- $i_h \leq r_{h-1}$ *and* i_{h-1} *is a c-candidate.*

Moreover, we say that i_h is a last c-candidate if it is a c-candidate and one of the following conditions hold:

- $h = k$, *or*
- $r_h < i_{h+1}$.

Intuitively, a position is a *c-candidate* if it is a c-run break and its LCP_r value is equal to the LCP_r value of every previous c-run break (if any) within its box. On the other hand, a *c-candidate* position is a *last c-candidate* if there are no larger *c-candidates* positions within its box. The following proposition establishes the equivalence between finding c-run break positions that are the rightmost maximum within a box and *last c-candidate* positions.

Proposition 1. *Let i_h be a c-run break. We have*

$$i_h = \max\{i \ : \ i \in B_{i_h,c} \wedge \mathrm{LCP}_r[i] = \max \mathrm{LCP}_r[B_{i_h,c}]\}$$

if and only if i_h is a last c-candidate.

Proof. Let $B_{i_h,c} = \{i_{h-a}, \ldots, i_h, \ldots, i_{h+b}\}$, $box(i_h) = [l_h, r_h]$, and define $i_0 := 0$ and $i_{k+1} := n+1$. By definition of $B_{i_h,c}$, for every $j \in [h-a, h+b]$ it holds $\mathrm{LCP}_r[i_h] \leq \mathrm{LCP}_r[i_j]$. In addition, it holds

$$i_{h-a-1} < l_h \leq i_{h-a} < \cdots < i_h < \cdots < i_{h+b} \leq r_h < i_{h+b+1} \tag{2}$$

($\Rightarrow$) Suppose $i_h = \max\{i \ : \ i \in B_{i_h,c} \wedge \mathrm{LCP}_r[i] = \max \mathrm{LCP}_r[B_{i_h,c}]\}$. By the *maximality* of $\mathrm{LCP}_r[i_h]$, for every $i_j \in B_{i_h,c}$ it holds $\mathrm{LCP}_r[i_h] = \mathrm{LCP}_r[i_j]$. This implies $b = 0$ and $box(i_{h-a}) = \cdots = box(i_h)$. Since by Eq. (2) it holds $i_{h-a-1} < l_{h-a} \leq i_{h-a}$, i_{h-a} is a *c-candidate*. In addition, by Eq. (2), for $d = 1, \ldots, a$ we have $i_{h-a+d} < r_{h-a+d-1}$, and by the equality of the boxes we have $r_{h-a+d-1} = r_h$, then i_{h-a+d} is a *c-candidate*, so i_h is a *c-candidate*. Finally, by Eq. (2) it holds $r_h < i_{h+1}$, therefore, i_h is a *last c-candidate*.
($\Leftarrow$) We proceed by contrapositive. Suppose $i_h \neq i_m = \max\{i \ : \ i \in B_{i_h,c} \wedge \mathrm{LCP}_r[i] = \max \mathrm{LCP}_r[B_{i_h,c}]\}$. If $m < h$, since $i_m \in B_{i_h,c}$, there exists $e \in [h-m]$ such that $\mathrm{LCP}_r[i_{h-e}] > \mathrm{LCP}_r[i_{h-e+1}] = \cdots = \mathrm{LCP}_r[i_h]$. Because $\mathrm{LCP}_r[i_{h-e}] > \mathrm{LCP}_r[i_{h-e+1}]$, we have $l_{h-e+1} \leq i_{h-e}$ and $r_{h-e} < i_{h-e+1}$, so i_{h-e+1} is not a *c-candidate*. Now, for $d = 1, \ldots, e$ we have $l_{h-e+d} \leq i_{h-e+d-1}$ and $i_{h-e+d-1}$ is not a *c-candidate*, then i_{h-e+d} is not a *c-candidate*, which implies i_h is not a *c-candidate*. On the other hand, if $h < m \leq k$, then $1 \leq b$ and $i_{h+1} \in B_{i_h,c}$, so we have $\mathrm{LCP}_r[i_h] \leq \mathrm{LCP}_r[i_{h+1}]$, which means $i_{h+1} \leq r_h$. Therefore, i_h cannot be a *last c-candidate*. □

From Proposition 1 we can develop a particular strategy to find local maxima inside *boxes* by finding *last candidate*s within them. Actually, since the *candidate* condition propagates forward, a single sequential scan of the BWT_r suffices to identify *last candidate* positions, given that we know boxes' boundaries. As mentioned in Section 2, we can compute $box(i) = [l_i, r_i] = [\mathrm{PSV}[i]+1, \mathrm{NSV}[i]-1]$ in $O(1)$ time.

Following the above ideas, we show a *linearization* of the quadratic algorithm in Algorithm 1. We call this algorithm *plain last-candidate* (PLC) algorithm (the reason for "plain" will be made clear later). It uses a data structure R to save information about the *candidate* condition on the last c-run break seen up to now. The algorithm sequentially scans the BWT_r looking for run-breaks. For a given c-run break i, where $\mathrm{BWT}_r[i'] = c$ for some $i' \in \{i-1, i\}$, $R[c]$ stores $(sa\_pos \leftarrow i, text\_pos \leftarrow n - \mathrm{SA}_r[i'] + 1, candidate \in \{true, false\})$, where *candidate* is set to *true* if and only if position i is a *c-candidate*.

Algorithm 1 shows the PLC algorithm. We next prove its correctness.

Lemma 1. *Given $T \in \Sigma^n$, Algorithm 1 computes a suffixient set of smallest cardinality $\mathcal{S}$ in $O(n)$ time and $O(n)$ space.*

Proof. For each $c \in \Sigma$, $R[c]$ is set to $(1, 0, true)$ at the beginning of the algorithm (line 4). Then, the first time we find a c-run break, the condition of line 9 is never satisfied (because for every c-run-break we initially set $R[c].sa\_pos = 1$ and $\mathrm{NSV}[1] = n+1$), but the condition of line 13 is true, so $R[c]$ is updated and set as a *c-candidate* (because the first c-run break is always a *c-candidate*).

Algorithm 1: PLC-Algorithm

input : A string $T[1,n]$ over a finite alphabet Σ.
output : A smallest suffixient set for T.

1 $\mathcal{S} \leftarrow \emptyset$;
2 $\text{BWT}_r \leftarrow \text{BWT}(T^{\text{rev}})$; $\text{LCP}_r \leftarrow \text{LCP}(T^{\text{rev}})$; $\text{SA}_r \leftarrow \text{SA}(T^{\text{rev}})$;
3 NSV $\leftarrow$ NSV(LCP); PSV $\leftarrow$ PSV(LCP);
4 $R[1,\sigma] \leftarrow (sa\_pos \leftarrow 1, text\_pos \leftarrow 0, candidate \leftarrow true) \times \sigma$;
5 **for** $i = 2, \ldots, n$ **do**
6 | **if** $\text{BWT}_r[i] \neq \text{BWT}_r[i-1]$ **then**
7 | | **for** $i' \in \{i-1, i\}$ **do**
8 | | | $candidate \leftarrow false$; $c \leftarrow \text{BWT}_r[i']$;
9 | | | **if** $\text{NSV}[R[c].sa\_pos] \leq i \wedge R[c].candidate = true$ **then**
10 | | | | $\mathcal{S} \leftarrow \mathcal{S} \cup \{R[c].text\_pos\}$;
11 | | | **else**
12 | | | | $candidate \leftarrow R[c].candidate$;
13 | | | **if** $R[c].sa\_pos \leq \text{PSV}[i]$ **then**
14 | | | | $candidate \leftarrow true$;
15 | | | $R[c] \leftarrow (i, n - \text{SA}_r[i'] + 1, candidate)$;
16 **foreach** $c \in \Sigma$ **do**
17 | **if** $R[c].candidate = true$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{R[c].text\_pos\}$;
18 **return** $\mathcal{S}$;

Now, assume that we have scanned the BWT_r until position $i-1$ and i is a c-run break, with $\text{BWT}_r[i'] = c$ for some $i' \in \{i-1, i\}$. Also, assume that $j < i$ is the previous c-run break, $j' \in \{j-1, j\}$ and $\text{BWT}_r[j'] = c$. Then $R[c].sa\_pos = j$, $R[c].text\_pos = n - \text{SA}_r[j'] + 1$. If $R[c].candidate = true$ and $\text{NSV}[R[c].sa\_pos] \leq i$ (line 9), then j is a *last c-candidate* and we add $R[c].text\_pos$ to the suffixient set we have built up to now. On the other hand, if $R[c].sa\_pos \leq \text{PSV}[i]$ or j is a *c-candidate* and $i < \text{NSV}[R[c].sa\_pos]$ (line 13) we set i as a *c-candidate* (lines 14 and 15), otherwise it is set as a non-*c-candidate* (lines 8 and 15). Now, we update the remaining attributes of $R[c]$ according to i's values: $R[c].sa\_pos = i$, $R[c].text\_pos = n - \text{SA}_r[i'] + 1$. Finally, if i is the last c-run break, it is added to the output suffixient set in the final loop if it is a *c-candidate* (line 16). Therefore, at the end of the algorithm execution, the output set $\mathcal{S}$ consists of all (and only the) *last candidate* positions. Then, Algorithm 1 correctness follows from correctness of the quadratic algorithm [6,9], and Proposition 1.

The arrays BWT_r, LCP_r, SA_r, NSV, and PSV consume $O(n)$ space. The data structure R uses $O(\sigma)$ space and the set $\mathcal{S}$ is also of size $O(n)$. Thus, the space usage of Algorithm 1 is $O(n)$. With regard to running time, computing BWT_r, LCP_r, SA_r in line 2 takes $O(n)$ time. PSV and NSV in line 3 can also be computed in $O(n)$ time [1]. Scanning BWT_r takes $O(n)$ time, and we perform $O(1)$ operations for each BWT_r position. The total running time is then bounded by $O(n)$. □

Cenzato et al. [5] give two linear-time algorithms to compute a smallest suffixient set. One is based on the LF array (so we refer to it as LF algorithm) and the other is based on PSV/NSV arrays (and we call it FM algorithm). As pointed out in the paper, the FM algorithm is faster than the LF algorithm, at the cost of increasing space usage. As shown in Table 1, the PLC algorithm is neither faster than the FM algorithm nor uses less space. Interestingly, with a slight modification, we can turn it as fast as the FM algorithm using as little space as the LF algorithm, getting the best performance of both. We refer to the new algorithm as LC.

By the form in which Algorithm 1 works, we can see that it never asks for PSV/NSV values at positions that are not run-breaks. So, there is no need to compute PSV/NSV for other positions. Further, we can compute the needed values *on the fly* as we find the run-breaks, using a stack-based strategy similar to the one of Bonizzoni et al. [2]. In particular, we use two stacks, psv and nsv. In psv, we push each i as we iterate from $i = 2$ to $i = n$, that is, every i can be the PSV value for some run-break. Then, for each run-break i we pop elements from psv until $\mathrm{LCP}_r[psv.top()] < \mathrm{LCP}_r[i]$ or psv is empty, so $\mathrm{PSV}[i] = psv.top()$ or $\mathrm{PSV}[i] = 1$, respectively. Since we compute each needed PSV value when it is requested, we store it in a variable psv_i, so there is no need to create the array PSV, which reduces running time and space usage. On the other hand, in nsv we push a c-run break j to mean it is still pending to find its NSV value, which will not be required until the next occurrence of a c-run break. At the beginning of iteration i, while $\mathrm{LCP}_r[i] < \mathrm{LCP}_r[nsv.top()]$ we set $R[\mathrm{BWT}_r[nsv.top()-1]].nsv = R[\mathrm{BWT}_r[nsv.top()]].nsv = i$ (because we had defined $R[\mathrm{BWT}_r[j-1]].sa\_pos = R[\mathrm{BWT}_r[j]].sa\_pos = j$ when we processed the run-break j) and call $nsv.pop()$ (because we already found $nsv.top()$ NSV value). Since we store NSV values only for the run-breaks, we can store those values in the data structure R, adding a new field $R[c].nsv$, which is set $R[c].nsv = n+1$ as a default value when we update $R[c]$ on line 15 until we compute its real value (which can be also $n+1$). Then there is also no need to create an array NSV, which again reduces running time and space usage. Algorithm 2 applies these modifications to Algorithm 1.

4 Experimental Results

In this section we empirically compare our new algorithm with previous ones that are implemented. We also implement and compare BGR algorithm [2].

The experiments were run using a workstation Intel(R) Pentium(R) IV, CPU @ 2.4 GHz with 8 cores, 10 MB cache, 256 gigabytes of RAM, 860 gigabytes of disk running Devuan GNU/LINUX 2.1. We compared the running time and the space consumption of algorithm LC against PLC, FM, and LF, computing smallest suffixient set over DNA sequences from the Pizza&Chili collection[3]. All the implementations of those algorithms are publicly available at https://

[3] https://pizzachili.dcc.uchile.cl

Algorithm 2: LC-Algorithm

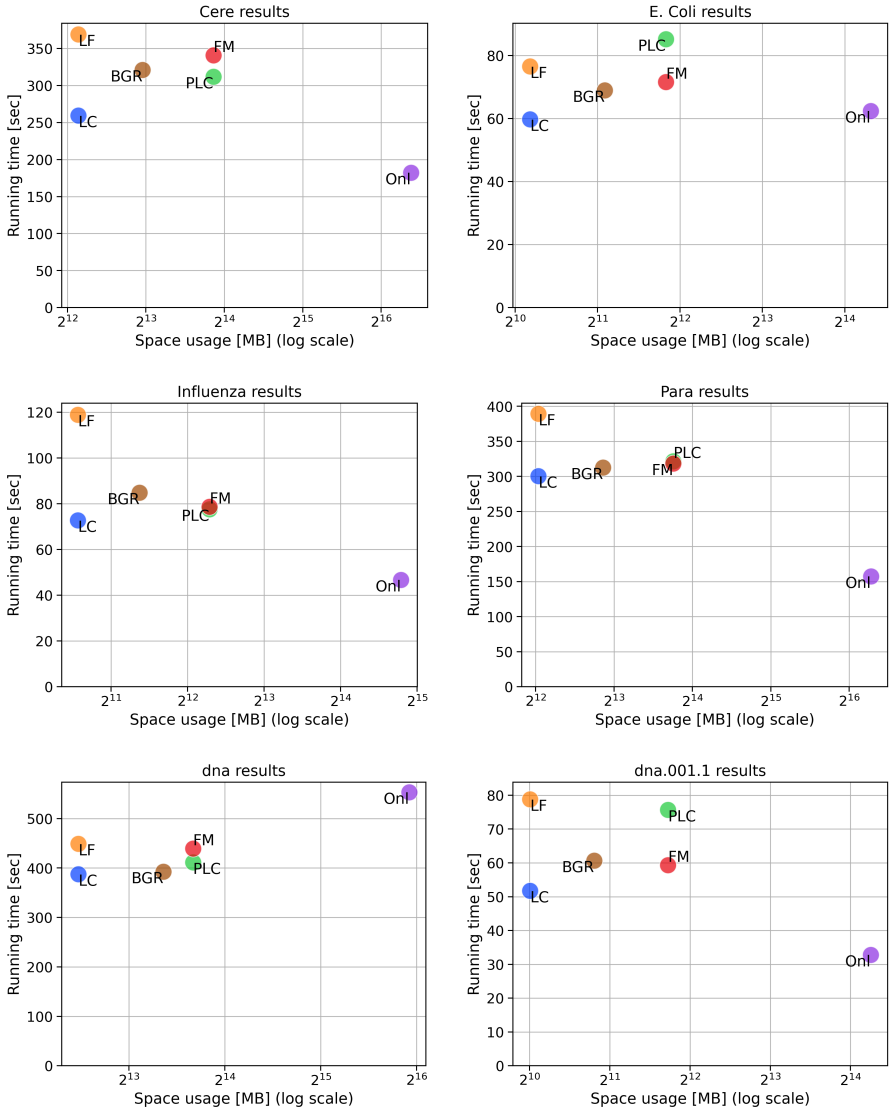
input : A string $T[1,n]$ over a finite alphabet Σ.
output : A smallest suffixient set for T.

$\mathcal{S} \leftarrow \emptyset$;
$\mathrm{BWT}_r \leftarrow \mathrm{BWT}(T^{\mathrm{rev}})$; $\mathrm{LCP}_r \leftarrow \mathrm{LCP}(T^{\mathrm{rev}})$; $\mathrm{SA}_r \leftarrow \mathrm{SA}(T^{\mathrm{rev}})$;
$R[1,\sigma] \leftarrow (sa\_pos \leftarrow 1, text\_pos \leftarrow 0, candidate \leftarrow true, nsv \leftarrow n+1) \times \sigma$;
$psv \leftarrow new\_stack()$; $nsv \leftarrow new\_stack()$; $psv.push(1)$; $nsv.push(1)$;
for $i = 2, \ldots, n$ **do**
  while *nsv is not empty and* $\mathrm{LCP}_r[i] < \mathrm{LCP}_r[nsv.top()]$ **do**
    for $j \in \{nsv.top() - 1, nsv.top()\}$ **do** $R[\mathrm{BWT}_r[j]].nsv \leftarrow i$;
    $nsv.pop()$;
  if $\mathrm{BWT}_r[i] \neq \mathrm{BWT}_r[i-1]$ **then**
    $nsv.push(i)$;
    while *psv is not empty and* $\mathrm{LCP}_r[i] \leq \mathrm{LCP}_r[psv.top()]$ **do** $psv.pop()$;
    if *psv is not empty* **then** $psv_i \leftarrow psv.top()$ **else** $psv_i \leftarrow 1$;
    for $i' \in \{i-1, i\}$ **do**
      $candidate \leftarrow false$; $c \leftarrow \mathrm{BWT}_r[i']$;
      if $R[c].nsv \leq i \wedge R[c].candidate = true$ **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{R[c].text\_pos\}$;
      else
        $candidate \leftarrow R[c].candidate$;
      if $R[c].sa\_pos \leq psv_i$ **then**
        $candidate \leftarrow true$;
      $R[c] \leftarrow (i, n - \mathrm{SA}_r[i'] + 1, candidate, n+1)$;
  $psv.push(i)$;
foreach $c \in \Sigma$ **do**
  if $R[c].candidate = true$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{R[c].text\_pos\}$;
return $\mathcal{S}$;

github.com/regindex/suffixient. We used the *elapsed time* and the *maximum resident set size* metrics returned by the utility `usr/bin/time`[4] as the running-time and the space consumption of the algorithms, respectively. We include in the construction cost the time to build the suffix array and other structures from the text. We also measure the performance of the algorithm of Bonizzoni et al. [2] implemented by us (called BGR, the implementation is available upon request) and the online algorithm [12] (called Onl, also available upon request), the latter as a way to compare a different paradigm to compute suffixient sets.

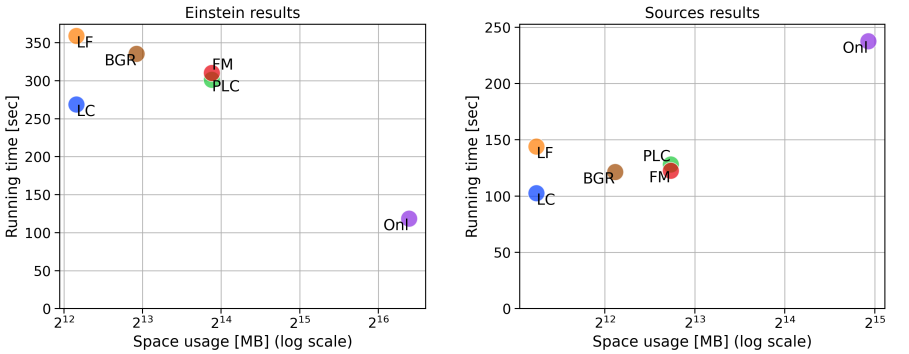
The results on DNA are shown in Figure 1. We can see that LC used the same space as LF, 1.7–1.9 times less space than BGR, 3.1–3.3 times less space than PLC and FM, and 17–19 times less space than Onl (except on dna, where the ratios are 2.3 and 11, respectively). On the other hand, LC was 14%–39% faster than LF, 1%–24% faster than BGR, 6%–32% faster than PLC, and 6%–24% faster than FM. Compared to Onl, LC gave mixed results, being from 30% faster to 48% slower. From those results, we conclude that LC outperforms all

[4] https://www.gnu.org/software/time/



			space (MB)						time (sec)					
Dataset T	length n	χ	**LC**	**LF**	**PLC**	**FM**	**BGR**	**Onl**	**LC**	**LF**	**PLC**	**FM**	**BGR**	**Onl**
Cere	461,286,644	9,921,698	4509	4509	14872	14873	7954	85336	259.55	368.95	311.72	340.68	321.02	182.15
E. Coli	112,689,515	13,119,749	1160	1160	3638	3638	2178	20371	59.74	76.58	85.20	71.60	68.96	62.38
Influenza	154,808,555	2,225,543	1516	1516	4995	4995	2651	28330	72.73	118.95	77.65	78.76	84.81	46.74
Para	429,265,758	13,382,184	4197	4196	13840	13841	7439	79548	300.61	389.34	321.62	318.45	312.66	157.63
dna	403,927,746	215,193,300	5686	5686	13024	13024	10523	62112	387.57	448.71	411.25	439.21	392.48	553.33
dna.001.1	104,857,600	1,414,698	1028	1028	3386	3386	1790	19500	51.74	78.84	75.66	59.32	60.65	32.78

Fig. 1. Space usage and running time comparison between all construction algorithms on DNA sequences from Pizza&Chili collection. Logscale in the space coordinate.



			space (MB)						time (sec)					
Dataset T	length n	χ	**LC**	**LF**	**PLC**	**FM**	**BGR**	**Onl**	**LC**	**LF**	**PLC**	**FM**	**BGR**	**Onl**
Einstein	467,626,544	200,245	4571	4571	15077	15077	7776	85854	268.68	359.12	301.32	310.3	335.43	118.46
Sources	210,866,607	36,693,523	2416	2416	6802	6802	4431	31159	102.54	143.96	128.1	122.57	121.47	237.51

Fig. 2. Space usage and running time comparison between all construction algorithms on **einstein.en.txt** and **sources** sequences from Pizza&Chili collection. Logscale in the space coordinate.

the other algorithms in *both* time and space, with the exception of the running time of Onl, where the comparison is mixed. It must be noted that the good time performance of Onl comes at the price of using a large amount of space (nearly 20 times that of LC) which is prohibitive in many practical settings.

Figure 2 shows the case of some non-DNA representative texts: natural language and source code. The results stay in the same line as on DNA. We can see that Onl performs noticeably better when χ is small, while the other constructions (including ours) are much more sensitive to n.

5 Conclusions and Future Work

We have proposed the first algorithm that is (i) one-pass (over the suffix-array domain), (ii) linear-time, and (iii) implemented, using the same space as the linear-time implementation using least space [5] *and* faster than the fastest implementation in that paper. Our algorithm also outperforms our implementation of that of Bonizzoni et al. [2] in time and space, and shows to be more practical than an online algorithm [12], in the sense that uses much less space (by a factor around 20), being faster on some inputs and slower on others. We do not know of other implementations.

As a future line of research, we plan to evaluate if algorithm FM [5] admits the same transformation we made on PLC to obtain LC: computing PSV and NSV values *on the fly* and just for run-breaks. That would yield a one-pass version of FM as well. We believe that, with that transformation, FM would be even faster than LC while not using more space.

Being one-pass opens the door to running the construction in streaming mode, where the suffix-array related structures of the reverse text are visited sequentially and not maintained in main memory. A relevant measure for such a streamed construction is the amount of main memory needed. Our algorithm needs only $O(\overline{r})$ main memory space for NSV because it is built only on run breaks, and $O(\sigma)$ for other structures, but it stores an amount of PSV array cells that is proportional to the height of the suffix tree. While small in practice, this height can be up to $\Theta(n)$ in pathologic cases. Reducing this extra space, say to $O(\overline{r})$ as well, would yield a streamed construction whose main memory usage is sensitive to the compressibility of the text.

We also plan to code LC (and eventually the one-pass version of FM) on top of PFP [3]. Cenzato et al. [5] experiment with running the algorithms over the concatenation of many copies of large DNA sequences, producing huge highly-repetitive files. In such context, the PFP version of the one-pass algorithm performed much better than any of the linear-time algorithms. We believe the PFP-based version of LC can be even faster in that setting.

References

1. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. Journal of Algorithms **14**(3), 344–370 (1993). https://doi.org/10.1006/jagm.1993.1018
2. Bonizzoni, P., Gao, Y., Riccardi, B.: Constructing Suffixient Arrays Revisited. In: Proc. 37th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 30:1–30:18 (2026). https://doi.org/10.4230/LIPIcs.CPM.2026.30
3. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big BWTs. Algorithms for Molecular Biology **14**(1), 13:1–13:15 (2019). https://doi.org/10.1186/S13015-019-0148-5
4. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
5. Cenzato, D., Depuydt, L., Gagie, T., Kim, S.H., Manzini, G., Olivares, F., Prezza, N.: Suffixient arrays: a new efficient suffix array compression technique. CoRR **abs/2407.18753** (2025). https://doi.org/10.48550/ARXIV.2407.18753
6. Cenzato, D., Olivares, F., Prezza, N.: On computing the smallest suffixient set. In: Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE). pp. 73–87 (2024). https://doi.org/10.1007/978-3-031-72200-4_6
7. Cenzato, D., Olivares, F., Prezza, N.: Testing suffixient sets (2025), https://arxiv.org/abs/2506.08225
8. Cobas, D., Gagie, T., Navarro, G.: Fast and small subsampled r-indexes. ACM Transactions on Algorithms **22**(1), article 7 (2025). https://doi.org/10.1145/3750729
9. 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
10. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4), 552–581 (2005). https://doi.org/10.1145/1082036.1082039
11. Fujimaru, H., Navarro, G., Olivares, F., Radoszewski, J., Romana, G., Urbina, C.: Computing smallest suffixient arrays in almost sublinear time. In: Proc. 33rd International Symposium on String Processing and Information Retrieval (SPIRE) (2026), to appear

12. Fujimaru, H., Navarro, G., Romana, G., Urbina, C.: Smallest suffixient sets: Effectiveness, resilience, and calculation (2026), https://arxiv.org/abs/2506.05638, to appear in *Theoretical Computer Science*
13. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. Journal of the ACM **67**(1), 2:1–2:54 (2020). https://doi.org/10.1145/3375890
14. Kempa, D., Kociumaka, T.: Breaking the O(n)-barrier in the construction of compressed suffix arrays and suffix trees. In: Proc. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 5122–5202. https://doi.org/10.1137/1.9781611977554.ch187
15. Köppl, D., Kucherov, G.: Smallest suffixient set maintenance in near-real-time. CoRR **abs/2604.27548** (2026), https://arxiv.org/abs/2604.27548, to appear in *MFCS'26*
16. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010). https://doi.org/10.1089/cmb.2009.0169
17. Navarro, G., Romana, G., Urbina, C.: Smallest suffixient sets as a repetitiveness measure. In: Proc. 32nd International Symposium on String Processing and Information Retrieval (SPIRE). pp. 217–232 (2025). https://doi.org/10.1007/978-3-032-05228-5_18