Theory of Computing Systems (2026) 70:50
https://doi.org/10.1007/s00224-026-10287-6

RESEARCH

Suffixient Arrays: A New Efficient Suffix Array Compression Technique

Davide Cenzato[1] · Lore Depuydt[2] · Travis Gagie[3] · Sung-Hwan Kim[1,4] · Giovanni Manzini[5] · Francisco Olivares[6,7] · Nicola Prezza[1]

Received: 17 March 2025 / Accepted: 29 June 2026
© The Author(s) 2026

Abstract
The Suffix Array is a classic text index enabling on-line pattern matching queries via simple binary search. The main drawback of the Suffix Array is that it takes linear space in the text's length, even when the text itself is extremely compressible. Several works in the literature showed that the Suffix Array can be compressed, but they all rely on complex succinct data structures which in practice tend to exhibit poor cache locality and thus significantly slow down queries. In this paper, we propose a new simple and very efficient solution to this problem by presenting the *Suffixient Array*: a tiny subset of the Suffix Array *sufficient* to locate on-line one pattern occurrence (in general, all its Maximal Exact Matches) via binary search, provided that random access to the text is available. We prove that: (i) the Suffixient Array length χ is a strong repetitiveness measure, (ii) unlike most existing repetition-aware indexes such as the r-index, our new index is efficient in the I/O model, and (iii) Suffixient Arrays can be computed in linear time and compressed working space. We show experimentally that, when using well-established compressed random access data structures on repetitive collections, the Suffixient Array **sA** is *simultaneously* (i) faster and orders of magnitude smaller than the Suffix Array and (ii) smaller and *one to two orders of magnitude faster* than the r-index. With an average pattern matching query time as low as 3.5 ns per character, our new index gets very close to the ultimate lower bound: the RAM throughput of our workstation (1.18 ns per character).

Keywords Suffixient set · Maximal exact match · Right-maximal substring · Text indexing · Compressed-space algorithms

1 Introduction and Overview of the Paper

In this paper, we propose a new simple and remarkably efficient solution to the well-studied problem of indexing compressed text for fast pattern matching queries [3, 4]. This problem is motivated by the constantly-increasing amount of repetitive data

This article is a journal extension of an article that appeared in the proceedings of SPIRE 2024 [1] and of the arXiv pre-print [2].

Extended author information available on the last page of the article

produced in several applicative scenarios. Bioinformatics is an excellent example of such a phenomenon: with the quick advancement of DNA sequencing technologies in the last decade, an enormous amount of genomic data can now be produced efficiently in terms of both time and cost (for instance, since the introduction of Next-Generation Sequencing, the cost of sequencing a Human genome dropped from 1 million to just 600 dollars in roughly a decade [5]). These technological improvements opened the *pangenome era*, in which typical analyses often involve the comparison of billions of sequence fragments with large collections of reference genomes [6, 7]. This *data* explosion, however, did not correspond to an *information* explosion. Since genomes of the same species are nearly-identical (a typical human genome, for instance, differs by no more than 0.2% from the reference genome sequence [9]), a collection of same-species genomes is usually extremely compressible. The goal of a compressed index is to pre-process such a collection into a small (compressed) data structure in order to speed up subsequent pattern matching queries. Importantly, such queries must be supported directly on the compressed data without decompressing it.

Compressed Indexed Pattern Matching: State of the Art

Suffix arrays [10–12] are a classical (uncompressed) solution to the indexed pattern matching problem. They use linear space in the text's length and are extremely simple: the Suffix Array SA is the lexicographically-sorted list of (the starting positions of) the text's suffixes. Pattern matching on SA can be performed via simple binary search combined with text extraction, needed to compare the sought pattern with the text's suffixes during binary search.

As mentioned above, however, linear space is not acceptable in modern big-data scenarios. Subsequent research therefore focused on techniques for compressing the Suffix Array and for supporting pattern matching on conceptually-different compressed text representations, such as Lempel-Ziv'77 (LZ77) [13] and grammar compression [14]. First research papers on compressed suffix arrays focused on entropy compression and exact pattern matching [15, 16]. Entropy, however, is a weak compressibility measure in the repetitive regime as it is based on symbol frequencies. As a consequence, the empirical entropy of a text T and a collection composed of, say, thousands of copies of T is the same (i.e., entropy is not sensitive to repetitions). Since then, a variety of indexes have been developed to capture and exploit the repetitiveness of the input data in different ways; see [3, 4] for an extensive survey on compressed indexes and repetitiveness measures. Notable works in this research direction include the run-length FM-index of Mäkinen and Navarro [17], which used space proportional to the number of runs r in the Burrows-Wheeler transform (BWT) of the text (where r is known to effectively capture repetitiveness in the text, see [3]) to count the number of occurrences of a pattern. Another key contribution is the r-index of Gagie et al. [18], which extended these functionalities to locate queries (that is, reporting the occurrences of the pattern in the indexed text) in the same asymptotic space. These indexes can be interpreted as techniques to compress the Suffix Array to space proportional to $O(r)$ words. Compared with the Suffix Array, however, they suffer from a big issue: being based on backward searching the Burrows-Wheeler transform and on complex compressed data structures for *rank*, *select*, and *predecessor* queries, they can no longer be considered as simple data structures and are notoriously not cache

efficient. As a matter of fact, searching a pattern of length m in those indexes has $\Omega(m)$ I/O complexity (i.e., cache misses). In practice (as we also investigate in this article), this translates to the fact that these indexes are orders of magnitude slower than suffix arrays. This has recently been mitigated by techniques based on relative Lempel-Ziv (RLZ) compression of the Suffix Array [19], which however incur a larger space usage.

Extensions to Approximate Pattern Matching

While these works focused on *exact* pattern matching, in real applications one is usually interested in finding *approximate* occurrences of the pattern under some meaningful string distance (e.g. Hamming or edit distance). For instance, for approximately matching short reads under the edit distance, the tool b-move [20] offers a lossless, repetition-aware (run-length-compressed BWT) solution. Another widely used approach to approximate pattern matching, effective for various types of reads and commonly employed in practice [21], is to locate *Maximal Exact Matches* (MEMs), and then extend them to full (approximate) occurrences of the pattern. Given a pattern P, a MEM is a maximal (that is, not extendible either to the left nor to the right) substring $P[i, j]$ occurring without errors in the collection. Bannai et al. [22] showed that one can augment the r-index to compute MEMs efficiently within the same $O(r)$ space bound on top of a random-access oracle on the text. Their algorithm has two phases. In the first phase, it scans the pattern from right to left to compute for every suffix of the pattern the position in the text of an occurrence of the longest prefix of that suffix that occurs anywhere in the text. Then it scans the pattern again, but this time from left to right, to compute the length of a longest match of each of its suffixes using a random access oracle and the positions stored in the first phase, from which MEMs can be reported immediately. Subsequently, an efficient construction algorithm for the data structure of Bannai et al. [22] was presented by Rossi et al. [23], who also presented a practical implementation. The two-phases algorithm of Bannai et al. [22] was later simplified into a one-pass (left-to-right scan) algorithm by Boucher et al. [24] which allows for processing patterns in the streaming model.

1.1 Our Contributions

In this paper we present a new Suffix Array compression technique that is *repetition-aware*, *simple*, and remarkably *efficient in practice*. More in detail, we introduce the *Suffixient Array* (sA): a sampling scheme of the Prefix Array[1] (the co-lexicographically-sorted list of text prefixes) with the following properties.

1. If random access is available on the text, then a sequence of simple binary searches on sA suffices to locate *on-line* one occurrence of the pattern P in the text (i.e., we return an occurrence of every prefix $P[1, i]$ as soon as $P[i]$ arrives) or, more in gen-

[1] Equivalently, the Suffix Array of the reversed text; for a formal definition of Prefix Array, see Definition 4. Of course, one can define the same concepts symmetrically as a sampling of the Suffix Array of the original text; we decided to sample the Prefix (rather than Suffix) Array in order to support *on-line* pattern matching (read next).

eral, all its MEMs. Using state-of-the-art compressed random access techniques, we obtain a fully-compressed index.
2. sA is compressed in the sense that its length χ is upper-bounded by $O(\bar{r})$, the number of equal-letter runs in the Burrows-Wheeler transform of the reversed text. Furthermore, χ does not depend on the alphabet order (while $\bar{r}$ does) and there exist infinite text families such that $\chi \in o(\bar{r})$. This is confirmed in practice: on a repetitive collection containing variants of Human chromosome 19, our index is smaller than the *toehold lemma* of the r-index (i.e., its subset for locating one pattern occurrence — about half the size of the full r-index).
3. Unlike most existing compressed indexes, pattern matching on our index is also efficient in the I/O model. This is confirmed in practice: our index is *one to two orders of magnitude faster* than the r-index [18]. We compare our query times with a hard lower bound — the time needed on our workstation to extract $|P|$ contiguous characters from a random position in a large text (which approaches the RAM throughput as $|P|$ increases) — and show that our index approaches this bound.

While our index can locate only one pattern occurrence, standard techniques [18, 25] can be used to list efficiently all other occurrences by enumerating Suffix Array values adjacent to the pattern occurrence. We do not enter in such details in this article and will describe this line of research in a future publication under preparation.

When the uncompressed text is used for random access, our technique can be directly compared with binary searching the Suffix (Prefix) Array. In order to obtain a self-index, however, we need to use a compressed text representation. A vast amount of literature has been devoted to supporting fast random access in compressed space, for example via Straight-Line programs [26], LZ77 compression [27], block trees [28], or relative Lempel-Ziv (RLZ) [29]. See [3] for a complete survey on the topic. Some of these data structures are also known to be very small and fast in practice [28, 29]. As a consequence, they can be used as a black-box in point (2) above, arguably yielding the first *simple* (and efficient, as we show experimentally) compressed self-index for repetitive sequences.

Suffixient Sets

To achieve this result, we start by introducing a new class of combinatorial objects of independent interest: *suffixient sets*. Let $T \in \Sigma^n$ be a text of length n. A substring $\alpha = T[i, j-1]$ of T is said to be *right-maximal* if both $\alpha \cdot a$ and $\alpha \cdot b$ appear in T for some characters $a \neq b$, or if α is a suffix of T. Consider any set $S \subseteq [n]$ such that the following holds: for every one-character right-extension $T[i, j]$ of every right-maximal substring $T[i, j-1]$ of T, there exists $x \in S$ such that $T[i, j]$ is a suffix of $T[1, x]$. We call a set S with this property *suffixient* (being *sufficient* to "capture" all right-extensions of right-maximal strings or, equivalently, "cover" paths starting at the root of the suffix tree of T and ending in the first character labeling every edge) and denote with χ the size of a smallest (not necessarily unique) suffixient set for T.

We proceed by exhibiting a suffixient set of cardinality $2\bar{r}$, where $\bar{r}$ is the number of runs in the Burrows-Wheeler transform of T reversed. In particular, this implies $\chi \leq 2\bar{r}$, showing that χ is a new meaningful repetitiveness measure. Observe that χ is independent of the alphabet ordering, while $\bar{r}$ is not. Since re-ordering the alphabet

may reduce asymptotically $\bar{r}$ [30], from the bound $\chi \leq 2\bar{r}$ we immediately obtain that there exist string families on which $\chi = o(\bar{r})$. In other words, χ is a better repetitiveness measure than $\bar{r}$, even though we leave open the problem of determining whether χ is *reachable* (i.e., if $O(\chi)$ words are always sufficient to store the text). We get however close to this goal: we prove that any suffixient set is a string attractor [31], which implies that $O(\chi \log(n/\chi))$ words of space are sufficient to store T.

Suffixient Arrays

We define a *Suffixient Array* sA of T to be any (not necessarily unique) suffixient set S of smallest cardinality χ, sorted according to the co-lexicographic order of the text prefixes represented by S.

A Suffixient Array sA can be used to locate one occurrence of a pattern $P[1, m]$ in T with the following simple strategy. Assume that P occurs in T and that the prefix $P[1, j]$ (starting with $j = 0$) is right-maximal in T. (i) Find (by binary search on sA and random access on T) a prefix $T[1, x]$, for $x \in$ sA, suffixed by $P[1, j + 1]$. (ii) Right-extend the match by comparing $T[x + 1, \ldots]$ and $P[j + 2, \ldots]$. This way, we either match until the end of P or we find a mismatch $T[x + 1 + t] \neq P[j + 2 + t]$, for some $t \geq 0$. In the former case, we are done. In the latter case, we know that if $P[1, j + 1 + t]$ occurs in T, then it is right-maximal in T so we can repeat steps (i) and (ii). Starting with the empty prefix of P, this procedure yields one occurrence of P in T and, as we show in this paper, can be easily extended to return all MEMs of P in T. While this procedure extracts from the text a number of characters proportional to $|P|^2$ in the worst case, we show that it actually triggers at most $O((1 + m/B) \cdot d \log \chi)$ I/O operations (i.e., cache misses), where B is the I/O block size (i.e., cache line) and $d \leq m$ is the node depth of P in the suffix tree of T. In many practical applications (for example, DNA alignment), $d \ll m$ holds (on uniform random texts, $d \in O(\log_\sigma n)$ with high probability).

We furthermore show that the running time can be made linear in $|P|$ also in the worst case by employing z-fast tries [32] and fast longest common prefix/suffix queries on T in compressed space.

Computing the Suffixient Array

We proceed with the following natural question: can the Suffixient Array sA be computed efficiently? To answer this question, we start by characterizing the smallest suffixient set. Intuitively, we show that a suffixient set is of smallest cardinality when it captures precisely all the χ one-character extensions of all right-maximal substrings of T being also left-maximal. We call such substrings of T *supermaximal extensions*. We then describe four algorithms solving the problem, all based on the Burrows-Wheeler transform (BWT), the Longest Common Prefix array (LCP), and the Suffix Array (SA) of T reversed. The first algorithm runs in quadratic time and is conceptually very simple. The other three algorithms are optimizations of the first one. The second algorithm runs in $O(n + \bar{r} \log \sigma)$ time and requires only one sequential pass on those three arrays. The third and fourth algorithms require random access on those arrays and run in optimal $O(n)$ time. While being slower than the latter two when $\bar{r} \in \omega(n/\log \sigma)$, the one-pass property of the second algorithm makes it ideal to be combined with Prefix Free Parsing (PFP) [33], a technique able to stream those three

arrays in compressed working space. We show experimentally that this combination of PFP with our one-pass algorithm can quickly compute a smallest suffixient set in compressed working space on massive repetitive text collections.

We conclude the paper with experimental results testing Suffixient Array on pattern matching queries.

2 Preliminaries

We use the following notation: $[i, j]$ for an interval indicating all values $\{i, i+1, \ldots, j\}$ (if $j < i$, then $[i, j] = \emptyset$), $[n]$ for $[1, n]$, $A[1, n] \in U^n$ for an array A of length n over universe U whose indices range from 1 to n, $A[i]$ for the i-th element of A, $A[i, j]$ for the *sub-array* $A[i] \cdots A[j]$ of A, where $1 \leq i, j \leq n$ (if $j < i$, then $A[i, j] = \epsilon$ is the empty array), and $|A|$ for the length of A. More in general, $A[a, b]$, with $a \leq b \in \mathbb{N}$, indicates that A is an array whose indices range from a to b: the first element of A is $A[a]$, the second $A[a + 1]$, until the last which is $A[b]$.

Let Σ be a finite ordered alphabet of size σ. A *string* of length n over alphabet Σ is denoted with $\alpha[1, n] \in \Sigma^n$. Since a string over Σ can be viewed as an array over universe Σ, we use the same notations defined above also for strings. When $A[1, n]$ refers to a string, a sub-array $A[i, j]$ is also called a substring, a prefix when $i = 1$, and a suffix when $j = n$.

Given two strings $\alpha, \beta \in \Sigma^*$, $LCS(\alpha, \beta)$ denotes the length of the longest common suffix between α and β. Symmetrically, $LCP(\alpha, \beta)$ denotes the length of the longest common prefix between α and β.

A *run* for a string $\alpha[1, n]$ is a substring $\alpha[i, j]$ containing only one distinct character and satisfying (i) $i = 1$ or $\alpha[i-1] \neq \alpha[i]$, and (ii) $j = n$ or $\alpha[j] \neq \alpha[j+1]$. We denote by $\text{runs}(\alpha)$ the number of runs of α. For instance, $runs(AAACCTAAGGTAA) = 7$.

A *text* T is a string ending with a special character $T[n] = \$ \in \Sigma$ appearing only in $T[n]$ and such that $\$ < a$ for every $a \in \Sigma \setminus \{\$\}$. This special character is needed (as customary in the literature) for defining structures such as the suffix tree and the Burrows-Wheeler transform in a concise way with intended functionalities (see below for their definition). Since we will use these structures also on the reversed text, to ensure that \$ always appears at the end of the processed text, we define the reverse T^{rev} of a text T as $T^{\text{rev}} = T[n-1]T[n-2] \cdots T[1]\$$. To avoid confusion and improve readability, we will use symbols T and P to denote the input text and sought pattern (as well as their substrings using notation $T[i, j]$ and $P[i, j]$), and symbols $\alpha, \beta, \ldots$ to denote general strings. When T is the input text of our algorithms, by the above assumptions it holds that $n = |T| \geq 2$ and T contains at least two distinct characters. Moreover, we assume $\sigma \leq n$. This is not a strong requirement since the alphabet can always be re-mapped to $[n]$ without changing the solutions to the problems considered in this paper. Notice that re-mapping the alphabet does however change the space complexity of our algorithms since one needs to store the map, using additional space proportional to $O(\sigma')$ words (for example, using a perfect hash function), where $\sigma' \leq n$ is the number of distinct characters appearing in the text. Since, however, we will show that $\sigma' \leq \chi$, where χ is the size of the Suffixient Array, such map will not change asymptotically our space usage.

The following notions will be frequently used throughout the paper.

Definition 1 (Right-maximal substring) For a string $\alpha[1, n]$, a substring $\alpha[i, j]$ of α is a *right-maximal substring* (or *repeat*) if either (i) $j = n$ or (ii) there exist at least two distinct characters $a, b \in \Sigma$ such that both $\alpha[i, j] \cdot a$ and $\alpha[i, j] \cdot b$ are substrings of α.

Observe that, by the above definition, the empty string ϵ is right-maximal.

Definition 2 (Maximal exact match (MEM)) For strings $\alpha[1, n]$ and $\beta[1, m]$, a triplet (i, j, ℓ) with $i \in [m]$, $j \in [n]$ and $\ell \in [\min\{n, m\}]$ is a *Maximal Exact Match (MEM)* of β with respect to α if and only if the following three conditions hold: (i) $\beta[i - \ell + 1, i] = \alpha[j - \ell + 1, j]$, (ii) $i - \ell + 1 = 1$ or $\beta[i - \ell, i]$ is not a substring of α, (iii) $i = m$ or $\beta[i - \ell + 1, i + 1]$ is not a substring of α.

The above definition of a MEM is also sometimes referred to as a *super-maximal exact match (SMEM)* [34] (not to be confused with the notion of *supermaximal extension* introduced later in Definition 33). Observe that we take i and j to be the *ending* positions of the matches. This is clearly equivalent to taking the *beginning* positions of the matches (as is common in the literature), and simplifies the presentation of our algorithms.

Given two strings α and β, we define the *lexicographic order* $<_{\text{lex}}$ on Σ^* as follows: $\beta <_{\text{lex}} \alpha$ if β is a proper prefix of α, or if there exists an index $j \in [\min\{|\alpha|, |\beta|\}]$ s.t. $\beta[j] < \alpha[j]$ and for all $i \in [j - 1]$, $\beta[i] = \alpha[i]$. The co-lexicographic order $<_{\text{colex}}$ is defined symmetrically: $\beta <_{\text{colex}} \alpha$ if β is a proper suffix of α, or if there exists an index $j \in [\min\{|\alpha|, |\beta|\}]$ s.t. $\beta[|\beta| - j + 1] < \alpha[|\alpha| - j + 1]$ and for all $i \in [j - 1]$, $\beta[|\beta| - i + 1] = \alpha[|\alpha| - i + 1]$.

Definition 3 (Suffix array (SA)[35]) Given a text $T[1, n]$, the *Suffix Array* $\text{SA}(T)$ of T (SA for brevity) is the permutation of $[n]$ such that $T[\text{SA}[i], n] <_{\text{lex}} T[\text{SA}[j], n]$ holds for all $i, j \in [n]$ with $i < j$.

Symmetrically, the prefix array sorts text prefixes co-lexicographically:

Definition 4 (Prefix array (PA)) Given a text $T[1, n]$, the *Prefix Array* $\text{PA}(T)$ of T (PA for brevity) is the permutation of $[n]$ such that $T[1, \text{PA}[i]] <_{\text{colex}} T[1, \text{PA}[j]]$ holds for all $i, j \in [n]$ with $i < j$.

Definition 5 (LCP array [35]) Given a text $T[1, n]$, the *longest common prefix* array $\text{LCP}(T)$ of T ($\text{LCP}[2, n]$ for brevity) is the length-$(n-1)$ integer array such that $\text{LCP}[i]$ stores the length of the longest common prefix between $T[\text{SA}[i], n]$ and $T[\text{SA}[i - 1], n]$, for all $i \in [2, n]$.

Since we will frequently refer to multiple elements of the LCP array, we use the notation $\text{LCP}[B]$ for a set $B \subseteq [n]$ to denote the set $\{\text{LCP}[i] \ i \in B\}$.

Definition 6 (Suffix tree (ST), [36]) The suffix tree of a text $T[1, n]$ is an edge-labeled rooted tree with n leaves numbered from 1 to n such that (i) each edge is labeled with a non-empty string, (ii) each internal node has at least two outgoing edges, (iii)

the labels of outgoing edges from the same node start with different characters, and (iv) the string obtained by concatenating the edge labels on the path from the root to the leaf node numbered SA[i] is the i-th smallest suffix $T[\mathrm{SA}[i], n]$ where SA is the Suffix Array of T.

For a node v in a suffix tree, the string obtained by concatenating the edge labels on the path from the root to node v is called *path label* of v. By definition, the path label α of every suffix tree node is a right-maximal substring of T. Vice-versa, for every right-maximal substring α of T there exists exactly one suffix tree node having α as path label. For a (not necessarily right-maximal) string α, the highest node whose path label is prefixed by α is said to be the *locus* of α. For two nodes u and v such that their path labels are α and $a\alpha$, respectively, for some character $a \in \Sigma$ and string $\alpha \in \Sigma^*$, we say u can be reached by the *suffix link* from v.

Definition 7 (Burrows-Wheeler transform (BWT), [37]) Given a text $T[1, n]$, the Burrows-Wheeler Transform of T, BWT(T), is a permutation of the characters of T defined by concatenating the characters preceding the lexicographically-sorted suffixes of T, i.e., $\mathrm{BWT}(T)[i] = T[\mathrm{SA}[i] - 1]$ where we define $T[0] = T[n]$ for brevity.

We indicate with $\bar{r}$ the number of runs in $\mathrm{BWT}(T^{\mathrm{rev}})$. For example, if $T =$ `BANANA$`, then $T^{\mathrm{rev}} =$ `ANANAB$` (as stated above, we define the reverse of a text so that $ still appears at the end) and $\mathrm{BWT}(T^{\mathrm{rev}}) =$ `BNN$AAA`, so $\bar{r} = 4$.

We say that a randomized algorithm returns the correct result *with high probability* (abbreviated *w.h.p.*) if the probability of failure is at most n^{-c} for any constant c fixed before the algorithm begins. Here, n is the size of the input (in our cases, n will be the length of the input text).

3 Suffixient Sets

We start by defining the combinatorial object at the core of this paper: *suffixient sets*. We give two equivalent definitions based on suffix trees and on right-maximal strings.

We say that a set $S \subseteq [n]$ of positions in T is *suffixient* if the suffixes of the prefixes of T ending at those positions are *sufficient* to "cover" the *suffix* tree of T in the way described in the following definition:

Definition 8 (Suffixient set; definition based on suffix trees) A *suffixient set* for a text $T[1, n]$ is a set S of numbers between 1 and n such that, for any edge descending from a node u to a node v in the suffix tree of T, there is an element $x \in S$ such that u's path label is a suffix of $T[1, x - 1]$ and $T[x]$ is the first character of (u, v)'s edge label.

Because of the correspondence between suffix tree nodes and right-maximal substrings, an equivalent way to define suffixient sets is in terms of right-maximal substrings of T:

Definition 9 (Suffixient set; definition based on right-maximal strings) A set $S \subseteq [n]$ is *suffixient* for a text $T[1, n]$ 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 $T[1, x]$.

The following lemma shows that there is always a suffixient set for T of cardinality at most $2\bar{r}$. We note that suffixient sets can be even smaller: for the text T in Fig. 1 it holds $\bar{r} = 9$ but $\{14, 20, 33, 35\}$ is still suffixient.

Lemma 10 *For a text $T[1, n]$, let* SA *and* BWT *denote the Suffix Array and the Burrows-Wheeler transform of T^{rev}. The set $\{n - SA[i] + 1 \ : \ i = 1 \ \vee \ i = n \ \vee \ \text{BWT}[i] \neq \text{BWT}[i-1] \ \vee \ \text{BWT}[i] \neq \text{BWT}[i+1], \ 1 \leq i \leq n\}$ of positions on T, corresponding to characters at the at most $2\bar{r}$ run boundaries in* BWT*, is suffixient for T.*

Proof Consider an edge (u, v) descending from a node u to a node v in the suffix tree of T. Let α be u's path label and c be the first character in (u, v)'s edge label. By the definition of the BWT, the occurrences of characters immediately following occurrences of α in T are consecutive in the BWT of the reverse of T. By the definition of suffix trees, v must have a sibling, so not all the characters immediately following occurrences of α in T are copies of c. Therefore, for some x such that $T[x]$ is at a run boundary in the BWT of the reverse of T, we have $T[x - |\alpha|, x] = \alpha\, c$. Hence, α is a suffix of $T[1, x - 1]$ and $T[x] = c$ and position x "covers" the suffix tree edge (u, v). □

The next lemma shows that any suffixient set S for T is also a string attractor [31] for T — that is, for any non-empty substring $T[i, j]$ of T, some occurrence $T[i', j']$ of $T[i, j]$ is such that $S \cap [i', j'] \neq \emptyset$.

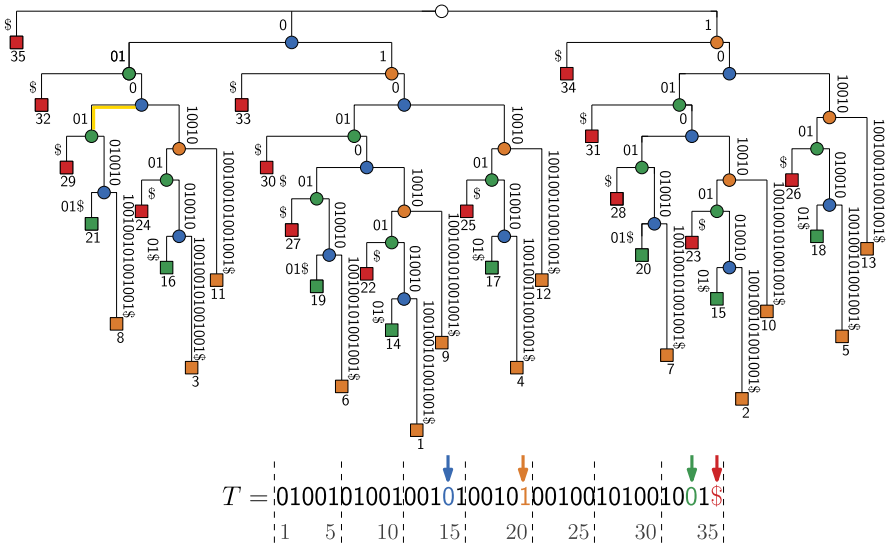

Fig. 1 A text T with highlighted the positions of the suffixient set $S = \{14, 20, 33, 35\}$. The figure also shows the suffix tree for T: each leaf is identified by the starting position in T of the associated suffix; for example, the third leaf from the left has index 29 since it corresponds to the suffix $T[29, 35] = 001001\$$. Node colors show how the set S covers the suffix tree: for each edge (u, v), the destination node v has the same color as the position in T covering it. For example, the parent of leaf 29 is colored in green since the edge (u, v) leading to that node (highlighted in yellow) is covered by $T[33]$. Indeed, u's path label is 0010 which is a suffix of $T[1, 32]$ and $T[33] = 0$ is the first character of (u, v)

Lemma 11 *Any suffixient set S for T is a string attractor for T.*

Proof Consider a non-empty substring $T[i, j]$ of T, let v be the locus of $T[i, j]$ in the suffix tree of T (that is, the highest node whose path label is prefixed by $T[i, j]$), let u be v's parent, let α be u's path label, and let c be the first character of (u, v)'s label. Then $\alpha\, c$ is a prefix of $T[i, j]$ and, since (u, v) is a single edge, any occurrence of $\alpha\, c$ in T is contained in an occurrence of $T[i, j]$. By Definition 9, there is some $x \in S$ such that $\alpha\, c$ is a suffix of $T[1, x]$, so $T[x]$ is contained in an occurrence of $\alpha\, c$ and thus contained in an occurrence of $T[i, j]$. □

As a consequence of the above results, if we denote the cardinality of the smallest suffixient set for T by χ, we have $\sigma' \leq \gamma \leq \chi \leq 2\bar{r}$, where σ' is the number of distinct characters appearing in T and γ is the size of the smallest string attractor for T. These relations show that χ is a new meaningful repetitiveness measure [3]. In addition, combining Lemma 11 with Kempa and Prezza's [31] results on string attractors, we immediately obtain the following:

Corollary 12 *Given a text T with smallest suffixient set of size χ, we can build a data structure of size $O(\chi \log(n/\chi))$ words of $O(\log n)$ bits supporting the extraction of any length-ℓ substring of T in $O(\log(n/\chi) + \ell)$ time.*

In particular, $O(\chi \log(n/\chi))$ words are sufficient to store the text. We leave open the problem of determining if $O(\chi)$ words are still sufficient. To conclude this section, note that χ is independent of the alphabet ordering, while $\bar{r}$ is not. Since re-ordering the alphabet may reduce asymptotically $\bar{r}$ [30], from the bound $\chi \leq 2\bar{r}$ we immediately obtain that there exist string families on which $\chi = o(\bar{r})$. In other words, on such strings χ is (asymptotically) a smaller repetitiveness measure than $\bar{r}$. Importantly note that, while it is true that the number of runs $\bar{r}$ of the BWT of T^{rev} can be minimized by choosing the best alphabet order, (the decisional version of) the problem of finding such an order is NP-complete [38]. On the other hand, in this paper we will show that a smallest suffixient set, of size χ, can be computed in linear time.

4 Suffixient Arrays: String Matching with Suffixient Sets

Based on the notion of suffixient set, we introduce *Suffixient Arrays*:

Definition 13 A *Suffixient Array* sA for a text T is an integer array of size χ whose entries form a suffixient set S of smallest cardinality for T, sorted by the co-lexicographic order of the text prefixes $T[1, x]$ corresponding to each $x \in S$.

Later, in Section 5 it will be made clear that a text T could admit multiple Suffixient Arrays. For the purposes of this section, any Suffixient Array will work.

Observe that the Suffixient Array is a sampling of the Prefix Array (Definition 4). In this article, we decided to sample the Prefix (rather than the Suffix) Array since, as we now show, this choice allows us to perform *on-line* pattern matching via a simple binary search strategy on sA. This feature is useful in streaming scenarios when long

patterns are provided one character at a time and one wishes to know an occurrence of each pattern prefix $P[1, i]$ immediately after $P[i]$ has arrived.

We first show in Section 4.1 how to use sA to find one occurrence of each pattern prefix $P[1, i]$ given in an on-line fashion, provided that random access on T is available. We analyze the running time of our algorithm in the worst-case scenario in the RAM and I/O models. Then, in Section 4.2 we show how to extend this idea to find all Maximal Exact Matches (MEMs) of P in T. To conclude, in Section 4.3 we speed up our queries by employing more advanced data structures (*z-fast tries* and *Straight-Line Programs*). Our strategies will actually work on *any* sorted suffixient set for T; however, since in Section 6 we show that a Suffixient Array sA can be computed in linear time and space (and even in compressed space), in the following we describe our algorithms using sA.

4.1 Locating a Single Pattern Occurrence

We present our solution in a general form in order to make it easier later (in Section 4.3) to plug in faster data structures to replace binary search. Let sA be a Suffixient Array for T. Assume that we have data structures on the pair (T, sA) supporting the following operations:

Definition 14 (random access: $T[i]$) Given an integer $i \in [n]$, return the i-th character of T.

Definition 15 (longest common suffix: $search(\alpha)$) Given a string $\alpha \in \Sigma^m$ with $m \geq 1$, return a pair $(x, \ell) \in \text{sA} \times \mathbb{N}$ such that (i) $\ell = LCS(\alpha, T[1, x])$ and (ii) there is no $x' \in \text{sA}$ with $LCS(\alpha, T[1, x']) > \ell$. If $\alpha[m] \neq T[x]$ for $\forall x \in \text{sA}$, then return the pair $(0, 0)$.

Observe that Operation $search(\alpha)$ can be supported in time $O(m \log \chi)$ by simple binary search on sA and random access on T, provided that random access on T can be performed in constant time per extracted character (otherwise, this time gets multiplied by time needed to access a single text character). This time can be sped up to $O(m + \log \chi)$ by using classic longest-common-suffix (LCS) data structures on the Suffixient Array (in [10], the authors show how to achieve this on the Suffix Array using the symmetric LCP array), using $O(\chi)$ additional words of space.

Algorithm 1 shows how to locate an occurrence of every $P[1, i]$, where P is a query pattern, using the above two operations.

Observe that, when operation $search()$ is implemented with binary search, Algorithm 1 works by performing a *sequence* of (at most) m simple binary searches. This is more than the single binary search of pattern matching on the Suffix (Prefix) Array, which however requires linear space in n (unless implemented with more complex compressed data structures such as the `rlcsa` [39]). Nevertheless, experimentally we will show that in practice binary searching sA is as fast as binary searching SA.

Lemma 16 *Algorithm 1 is correct and complete.*

Proof The claim is equivalent to the following:

Algorithm 1 Finding one occurrence of every prefix of $P[1, m]$ in $T[1, n]$.

input : A pattern $P[1, m]$.
output: For every $1 \leq i \leq m$, a pair (i, j) such that $T[j - i + 1, j] = P[1, i]$ if $P[1, i]$ occurs in T. Otherwise, terminates with NOT_FOUND at the smallest i such that $P[1, i]$ does not occur in T.

```
1 i ← j ← 0;
2 while i < m do
3 |   i ← i + 1; j ← j + 1;
4 |   if P[i] ≠ T[j] then
5 |   |   (j, ℓ) ← search(P[1, i]);
6 |   |   if ℓ < i then exit(NOT_FOUND) ;
7 |   end
8 |   output (i, j);
9 end
```

Proposition 17 *For every $1 \leq i \leq m$ the following hold. Every time Algorithm 1 reaches Line 8 with values i, j, $P[1, i]$ is a suffix of $T[1, j]$ so, in particular, $P[1, i]$ occurs in T. Conversely, if $P[1, i]$ occurs in T then Algorithm 1 will reach Line 8 with values i, j such that $P[1, i]$ is a suffix of $T[1, j]$.*

We prove the claim inductively on i. At the beginning of execution, in Line 3 we set $i \leftarrow j \leftarrow 1$. Assume $P[i]$ occurs in T. If $P[i] = T[j]$, then the `if` statement is not executed, we reach Line 8, and clearly Proposition 17 holds true. If, on the other hand, $P[i] \neq T[j]$, then the call $j \leftarrow search(P[1, i])$ at Line 5 will return a position j with $P[i] = T[j]$ since $P[i]$ is a one-character extension of a right maximal string (the empty string) and sA is a suffixient set. We conclude that, again, we reach Line 8 and Proposition 17 holds true. Assume now that $P[i]$ does not occur in T. Then, it must be the case $P[i] \neq T[j]$ so the `if` statement will be executed and the call $j \leftarrow search(P[1, i])$ will set j to 0, causing the termination of the algorithm with message NOT_FOUND. Also in this case, Proposition 17 holds true.

Assume inductively that $P[1, i - 1]$, with $i > 1$, occurs in T (otherwise, the algorithm halted in a previous step) and that the claim holds for all prefixes of $P[1, i - 1]$. Let $(i - 1, j - 1)$ be the pair output by the algorithm on prefix $P[1, i - 1]$. We consider two cases. (a) $P[1, i]$ occurs in T. Then, either (a.1) $P[i] = T[j]$, in which case the `if` statement is not executed, we reach Line 8, and Proposition 17 holds true, or (a.2) $P[i] \neq T[j]$. In this case, observe that $P[1, i - 1]$ is right-maximal in T since both $P[1, i]$ and $P[1, i - 1] \cdot T[j]$ occur in T. Then, the call $j \leftarrow search(P[1, i])$ at Line 5 will return a position j with $P[1, i] = T[j - i + 1, j]$ since $P[1, i]$ is a one-character extension of a right maximal string and sA is a suffixient set. We conclude that, again, we reach Line 8 and Proposition 17 holds true. (b) $P[1, i]$ does not occur in T. Then, necessarily $P[i] \neq T[j]$, so the `if` statement will be executed and the call $j \leftarrow search(P[1, i])$ will set j to 0, causing the termination of the algorithm with message NOT_FOUND. Also in this case, Proposition 17 holds true. □

Algorithm 1 calls $search(P[1, i])$ at most m times. We can say more: it is actually not hard to see that Algorithm 1 calls $search(P[1, i])$ at most $d \leq m$ times, where d is the node depth of the locus of P in the suffix tree of T. If Algorithm 1 outputs NOT_FOUND, then d is 1 more than the node depth of the locus of the longest prefix

of P that occurs in T. We conclude that, if $search(P[1, i])$ is implemented via binary search on sA and if random access on T costs constant time per extracted character, then Algorithm 1 runs in $O(m \cdot d \log \chi) \subseteq O(m^2 \log \chi)$ time. More in general we obtain:

Theorem 18 *Let $T[1, n]$ be a text stored in a data structure supporting the extraction of $T[i]$, for any $i \in [n]$, in time t_a. Then, given a pattern $P[1, m]$, using the Suffixient Array of T (χ words) we can locate one occurrence of every prefix $P[1, j]$ ($j \in [m]$) in total $O(t_a \cdot m \cdot d \log \chi) \subseteq O(t_a \cdot m^2 \log \chi)$ time, where $d \leq m$ is the node depth of the locus of P in the suffix tree of T.*

The above query time can be reduced to $O(t_a \cdot m \cdot d + d \log \chi)$ by using classic longest-common-suffix (LCS) data structures taking $O(\chi)$ additional words of space (see [10]). For random strings the length of the longest repeated substring is $O(\log_\sigma n)$ with high probability, which implies $d \in O(\log_\sigma n)$. We leave to future experiments the study of d in practical applications. However, the main feature of our solution is that, differently from the FM-index [15] and the r-index [18], our index is cache-efficient, provided that the text is stored contiguously in memory. While this seems a hard requirement for a *compressed* random access oracle, in Section 8 we will show such an efficient oracle. We analyze this phenomenon in the I/O model where a block (e.g. a cache line) fits B characters:

Theorem 19 *Let $T[1, n]$ be a text stored contiguously in main memory. Then, given a pattern $P[1, m]$, using the Suffixient Array of T (χ words) we can locate one occurrence of P with $O((1 + m/B) \cdot d \log \chi)$ I/O complexity, where $d \leq m$ is the node depth of the locus of P in the suffix tree of T.*

Proof As observed above, Algorithm 1 runs at most d binary searches, totaling $O(d \log \chi)$ binary search steps. At each step we need to compare at most m consecutive characters of P and T, an operation having $O(1 + m/B)$ I/O complexity. □

In comparison, the FM-index [15] and the r-index [18] have $\Omega(m)$ I/O complexity (i.e., every pattern character triggers a I/O operation). In Section 7 we will discuss heuristics that in practice tend to drastically reduce the impact of the term $d \log \chi$ of Theorem 19, effectively bringing the I/O complexity very close to the optimum.

Remark 20 (Locating all occurrences) As mentioned in the introduction, with our mechanism it is actually possible to count and locate *all* pattern occurrences (rather than just one) efficiently in the I/O model using $O(\bar{r})$ space on top of the random access oracle. This line of research, however, is out of the scope of this article and will be described in a future publication under preparation.

4.2 Finding Maximal Exact Matches

Algorithm 1 can be extended so that it returns all MEMs of P in T within the same time complexity. We present this procedure in Algorithm 2.

Algorithm 2 Finding all MEMs of $P[1, m]$ in $T[1, n]$.

```
   output: All MEMs (i, j, ℓ) of P with respect to T
1  i ← j ← 1;
2  ℓ ← 0;
3  while i ≤ m do
4  |  (j', ℓ') ← search(P[i − ℓ, i]);
5  |  if ℓ' < ℓ + 1 and ℓ > 0 then report (i − 1, j − 1, ℓ);
6  |  Δ ← LCP(P[i + 1, m], T[j' + 1, n]);
7  |  (i, j, ℓ) ← (i + Δ + 1, j' + Δ + 1, ℓ' + Δ);
8  end
9  report (i − 1, j − 1, ℓ);
```

If $P[i]$ does not occur in T, then Algorithm 2 simply does not output any triple of the form $(i, \cdot, \cdot)$. Assuming some oracle for functions $search(\cdot)$ and $LCP(\cdot, \cdot)$, we prove:

Lemma 21 *Given a pattern $P[1, m]$, Algorithm 2 reports all the MEMs of P in T.*

Proof The claim is implied by the following invariant. Every time the condition of the `while` statement (Line 3) is evaluated, the following Properties hold:

1. $\ell = LCS(T[1, j-1], P[1, i-1])$,
2. if $\ell > 0$ and $i \leq m$, then $P[i] \neq T[j]$,
3. $LCS(T[1, j'-1], P[1, i-1]) \leq \ell$ for every $j' \in [n]$, and
4. All MEMs (i', j', ℓ') such that $i' < i - 1$ have been reported.

The invariant trivially holds the first time we enter in the `while` loop.

We proceed by induction. Assume that the invariant holds at Line 3 for some values of j, ℓ, and $i \leq m$. We want to show that it still holds after executing the operations in Lines 4-7. First observe that, by Property (1) of the invariant, $search(P[i - \ell, i]) = search(P[1, i])$ holds in Line 4. This is a first difference with Algorithm 1, which instead calls $search(P[1, i])$ (we will need the variant of Line 4 in the next subsection).

Line does not modify i, j, ℓ, so it maintains the invariant valid. After running Line 4, by definition of $search()$ we have that $P[i - \ell' + 1, i] = T[j' - \ell' + 1, j']$ is the longest string (of length ℓ') suffixing a text prefix $T[1, x]$ ending in a position $x \in$ sA (in particular, $x = j'$ holds in this case). Observe also that $\ell' \leq \ell + 1$ because ℓ' cannot exceed the length of $P[i - \ell, i]$, which is $\ell + 1$. Moreover, in Line 5 we do not report any MEM with $\ell = 0$, so we can assume that $\ell > 0$ to analyze which MEMs are reported in Line 5. We have therefore to only distinguish two cases in Line 5.

(Case A) If $\ell' = \ell + 1$, then $(i - 1, j - 1, \ell)$ is not a MEM (because $P[i - \ell, i] = T[j' - \ell, j']$, i.e., $P[i - \ell, i - 1]$ can be extended to the right), and we correctly do not output it.

(Case B) If, on the other hand, $\ell' < \ell + 1$ then we claim that $(i - 1, j - 1, \ell)$ is a MEM (and we correctly report it). To see this, observe that (i) $P[i - \ell, i - 1]$ cannot be extended to the left (i.e., $P[i - \ell - 1, i - 1]$ does not occur in T). This is implied by Property (3) of the invariant. (ii) $P[i - \ell, i - 1]$ cannot be extended to the right (i.e., $P[i - \ell, i]$ does not occur in T). To see this, assume for a contradiction that $P[i - \ell, i]$ occurs in T. Let $\alpha = P[i - \ell, i - 1]$. Then, property (2) of the invariant

implies that α is right maximal, since both $\alpha \cdot P[i]$ and $\alpha \cdot T[j]$ occur in the text (with $P[i] \neq T[j]$). But then, by definition of suffixient set, there must exist $j'' \in$ sA such that $\alpha \cdot P[i] = P[i-\ell, i]$ (of length $\ell+1$) suffixes $T[1, j'']$. This contradicts the fact that $search(P[i-\ell, i])$ returned (j', ℓ') with $\ell' < \ell+1$. From (i) and (ii), we conclude that $(i-1, j-1, \ell)$ is a MEM so Line 5 correctly reports it.

Note that the above discussion implies that the algorithm correctly reported all MEMs ending before pattern position $i-1$ included.

At this point, we prove that $P[i-\ell'+1, i]$ cannot be extended to the left, i.e., that $P[i-\ell', i]$ does not occur in T. Assume, for a contradiction, that $P[i-\ell', i]$ (of length $\ell'+1$) occurs in T and let j'' be such that $P[i-\ell', i] = T[j''-\ell', j'']$. Recall that $\ell' \leq \ell+1$ must hold. If $\ell' = \ell+1$, then $LCS(T[1, j''-1], P[1, i-1]) \geq \ell' = \ell+1$, contradicting Property (3) of the invariant. If $\ell' \leq \ell$, then $T[j''-\ell', j''-1]$ (of length ℓ') is a suffix of $T[j-\ell, j-1]$, i.e., $T[j''-\ell', j''-1] = T[j-\ell', j-1] = P[i-\ell', i-1]$. But then $\alpha = P[i-\ell', i-1]$ is right-maximal: both $\alpha \cdot P[i]$ and $\alpha \cdot T[j]$ (with $P[i] \neq T[j]$ by Property (2) of the invariant), of length $\ell'+1$, occur in T. Again, by definition of suffixient set, this contradicts the fact that $search(i-\ell, i)$ returned (ℓ', j').

To conclude, let $\Delta = LCP(P[i+1, m], T[j'+1, n])$ as per Line 6. By definition of $LCP()$: (i) Since above we proved that $P[i-\ell'+1, i]$ cannot be extended to the left, it follows that the same holds for $P[i-\ell'+1, i+\Delta]$, (ii) $P[i+\Delta+1] \neq T[j'+\Delta+1]$ (importantly, note that $j'+\Delta+1 \leq n$ because of the terminator \$ at the end of T; $i+\Delta+1$, on the other hand, can be equal to $m+1$: in that case, we exit the `while` loop), (iii) $P[i-\ell'+1, i+\Delta] = T[j'-\ell'+1, j'+\Delta]$, and (iv) $P[i-\ell'+1, i+\delta]$ does not correspond to a MEM for all $0 \leq \delta < \Delta$. Conditions (i–iv) immediately imply the invariant is still valid on i, j, ℓ after the assignment at Line 7: Properties (1) and (3) of the invariant follow from (i) and (iii), Property (2) is equivalent to (ii), and property (4) follows from (iv) and from the fact that, as proved above, the algorithm correctly reported all MEMs ending before pattern position $i-1$ included. □

Implementing $search(\cdot)$ by simple binary search on sA and random access on T, and $LCP(\alpha, T[i, n]) = \Delta$ with a sequence of $\Delta+1$ random access queries on T, we obtain:

Theorem 22 *Let $T[1, n]$ be a text stored in a data structure supporting the extraction of $T[i]$, for any $i \in [n]$, in time t_a. Then, given a pattern $P[1, m]$, using the Suffixient Array of T (χ words) we can find all MEMs of P in T in total $O(t_a \cdot m^2 \log \chi)$ time.*

Proof Variable i in Algorithm 2 increases at least by one unit at every iteration of the `while` loop, and the loop terminates when $i > m$. In every iteration, we perform one call to $search()$ ($O(t_a \cdot m \log \chi)$ time with binary search on sA and random access on T), so overall all the calls to this operation cost $O(t_a \cdot m^2 \log \chi)$ time. Finally, operation $LCP(P[i+1, m], T[j'+1, n]) = \Delta$ implemented via random access costs $O(t_a \cdot \Delta)$ time. Since i is incremented by $\Delta+1$ in each step (until surpassing value m), it follows that the total time spent on $LCP()$ is $O(t_a \cdot m)$. □

4.3 Faster Queries

Using state-of-the-art techniques, our query times can be reduced from quadratic to linearithmic. Since a solution for MEMs also implies a solution for exact pattern matching, in this subsection we only discuss solutions for MEMs. We need two ingredients: z-fast tries [32] (in the revisited version of [40][2]) and Straight-Line Programs.

Lemma 23 *(*z-fast trie, *[40, Thm. 6-7]) Given a set $Z = \{\alpha_1, \cdots, \alpha_q\}$ of q strings sorted in co-lexicographic order ($i < j$ implies $\alpha_i <_{colex} \alpha_j$), there exists a data structure of $O(q)$ words of space that answers the following query: given a string $P[1, m]$, after an $O(m)$-time pre-processing of P, for any given substring β of P return the index i of the string $\alpha_i \in Z$ that has the longest common suffix with β and the length ℓ of this suffix (or any index $i \in [q]$ if $\ell = 0$). The query is answered in $O(\log m)$ time for each such queried substring β of P (after the linear pre-processing). The result is guaranteed to be correct if β is a substring of α_j for some $j \in [q]$.*

The last condition of Lemma 23 is needed because the z-fast trie data structure [40, Thm. 6-7] is based on hashing, and at construction time the best one can do is to build a hash function that is guaranteed to be perfect (collision-free) only on substrings of the input. Otherwise, the structure fails with low probability only [40, Thm. 7].

A Straight-Line Program (SLP) for a text T is a Context-Free Grammar generating only T and whose right-expansions are either one alphabet symbol or the concatenation of two nonterminals [26]. While we assume the reader to be familiar with the basics of grammar compression, we also refer to a survey [3, Sec. 3.2 & 4.1] for further reference. Nevertheless, for the purposes of this section, it is only important to know that the size (number of rules) g of a smallest SLP for T is a good measure of repetitiveness (see [3]). We use the following two SLP-based structures:

Lemma 24 *(*random access on SLPs, *[26]) Given an SLP with g rules for $T[1, n]$, there exists an $O(g)$-space (memory words) data structure answering random access queries $T[i]$ in $O(\log n)$ time.*

Lemma 25 *(*LCP/LCS queries on SLPs, *[41, Lemma 1]) Given an SLP with g rules for $T[1, n]$, there exists an $O(g)$-space (memory words) data structure with which we can preprocess any pattern $P[1, m]$ in $O(m)$ time such that later, given i, j and q, we can return $LCS(P[i, j], T[1, q])$ (or the symmetric $LCP(P[i, j], T[q, n])$ function) in $O(\log n)$ time [3] and with no chance of error as long as $P[i, j]$ occurs somewhere in T.*

Again, in Lemma 25 the requirement that $P[i, j]$ occurs somewhere in T is needed because the data structure uses hashing to compare strings, and perfect hashing is guaranteed only on T's substrings. Combining Lemma 23 and Lemma 25, we obtain the following lemma. Observe that the claim requires only $\beta[1, k-1]$ (not the full

[2] Although [40, Thm. 6] states a query time of $O(\log m + \sigma)$, below the theorem's statement the authors briefly observe that, within the same asymptotic space, query time can be reduced to $O(\log m)$.

[3] While in [41, Lemma 1] they only discuss $LCP()$, the symmetric function $LCS()$ can be supported by simply reversing the text, an operation that does not change the SLP size g.

queried string β) to be a substring of T. This will be crucial for the correctness of our MEM-finding algorithm.

Lemma 26 *Given a text $T[1,n]$ for which there exists a SLP of size g and given a Suffixient Array $\mathrm{sA}[1,\chi]$ for T, there exists a data structure of $O(\chi+g)$ words of space returning $(x,\ell) = search(\beta)$ on (T,sA) (Definition 15) in $O(\log n)$ time, for any string $\beta[1,k]$. The result is guaranteed to be correct if $\beta[1,k-1]$ is a substring of T.*

Proof We partition sA into the σ sub-arrays $\mathrm{sA}_a = \langle i-1 : i \in \mathrm{sA} \wedge T[i] = a\rangle$, for all $a \in \Sigma$, containing the text positions $i-1$ such that $i \in \mathrm{sA}$ and $T[i] = a$. Then, for all $a \in \Sigma$ we build the z-fast trie of Lemma 23 on the set of strings $Z_a = \{T[1,j] \;\; j \in \mathrm{sA}_a\}$. Letting $\sigma' \leq \chi$ denote the effective alphabet size (number of distinct characters appearing in T), these z-fast tries can be stored in an array of size $\sigma' \leq \chi$, whose indexes can be computed from the alphabet's characters via a perfect hash function ($O(\sigma') \subseteq O(\chi)$ space). This way, given $a \in \Sigma$ we can access the z-fast trie for Z_a in $O(1)$ time. Importantly, we build these z-fast tries so that they share the same hash function, constructed so as to be collision-free on the substrings of T. This implies that the z-fast tries return correct answers, provided that they are queried with substrings of T. We also build the LCS structure of Lemma 25 on T.

Recall that the z-fast trie for Z_a returns indexes i in the sorted Z_a. Let i be the index for $T[1,j]$ in the z-fast trie for Z_a. By using $O(\chi)$ additional words (e.g. a perfect hash function) we associate each such pair (i,a) to the corresponding text position j. In the following we will therefore assume that the z-fast tries return such text positions j (rather than indexes in Z_a).

To find $(x,\ell) = search(\beta)$, we proceed as follows. Note that $\beta[k] \neq T[j]$ for all $j \in \mathrm{sA}$ if and only if $\beta[k]$ does not appear in T (because sA is a string attractor). We can easily discover this event using an $O(\chi)$-space perfect hash function storing all distinct text's characters. In this case, we return the pair $(x,\ell) = (0,0)$.

Let us assume therefore that $\beta[k] = T[j]$ for some $j \in \mathrm{sA}$. We query the z-fast trie on $Z_{\beta[k]}$ with string $\beta[1,k-1]$. Since $\beta[1,k-1]$ is guaranteed to be a substring of T, the z-fast trie correctly returns position $j \in \mathrm{sA}_{\beta[k]}$ such that (i) j maximizes $LCS(T[1,j],\beta[1,k-1])$ among all positions in $\mathrm{sA}_{\beta[k]}$, and (ii) $T[j+1] = \beta[k]$. But then, this implies that $j+1 \in \mathrm{sA}$ maximizes $LCS(T[1,j+1],\beta[1,k])$ among all positions in sA.

After finding such $j+1 \in \mathrm{sA}$, we compute $\ell - 1 = LCS(T[1,j],\beta[1,k-1])$ using Lemma 25. Again, the structure is guaranteed to return the correct answer since $\beta[1,k-1]$ is a substring of T. To conclude, the result of the query is the pair (x,ℓ) with $x = j+1$. □

Implementing $LCP(\alpha, T[i,n]) = \Delta$ with a sequence of $\Delta+1$ random access queries on T, we obtain:

Theorem 27 *Let $T[1,n]$ be a text with Suffixient Array size χ for which there exists a SLP of size g. Then Algorithm 2, when implemented with the structure of Lemma 26 to support operation $search()$ and with the random access structure of Lemma 24 to support $LCP()$, finds all MEMs of any pattern $P[1,m]$ in T in $O(m \log n)$ time using $O(\chi+g)$ words of space. The algorithm always returns the correct answer.*

Proof Variable i in Algorithm 2 increases at least by one unit at every iteration of the `while` loop, and the loop terminates when $i > m$. In every iteration, we perform one call to $search()$ ($O(\log n)$ time with Lemma 26), so overall all the calls to this operation cost $O(m \log n)$ time. Finally, operation $LCP(P[i+1, m], T[j'+1, n]) = \Delta$ implemented via random access costs $O(\Delta \log n)$ time using the structure of Lemma 24. Since i is incremented by $\Delta + 1$ in each step (until surpassing value m), it follows that the total time spent on $LCP()$ is also $O(m \log n)$.

We now prove correctness. If the structure of Lemma 26 always returns the correct result, then the algorithm is correct by Lemma 21. Note that condition (1) of the invariant used in the proof of Lemma 21 implies that, in Line 4 of Algorithm 2, $P[i-\ell, i-1]$ is always a substring of T. But then, letting $\beta = P[i-\ell, i]$ and $k = |\beta|$, this means that all calls to $search$ are of the form $search(\beta)$, where $\beta[1, k-1]$ appears in T. This is precisely the condition required by Lemma 26 in order for the structure to return correct results. This concludes the proof. □

Remark 28 As far as the construction time is concerned, one can build the data structures described in this section in $O(n \log n)$ expected time where n is the text length, provided that an SLP with g rules is given. Computing a smallest suffixient set of size χ takes $O(n)$ time as we will see later in Section 6. The z-fast trie in Lemma 23 for the prefixes ending at positions in the suffixient set can be constructed in $O(n)$ expected time [40]. More specifically, one can build the enhanced suffix tree of the reversed text and Karp-Rabin fingerprints in $O(n)$ time, then insert the subset of the suffixes corresponding to the suffixient set in $O(\chi) \subseteq O(n)$ time with the computation of relevant information according to [40, Section 4]; if a collision is found, the procedure is repeated with a different hash function, and the expected number of repeats is $O(1)$. The random-access data structure in Lemma 24 can be built in $O(n)$ time [26, Theorem 1.1]. The construction time for the data structure in Lemma 25 is $O(n \log n)$ expected time [41, Lemma 1] whose bottleneck is to find a collision-free Karp-Rabin hash function relying on [42, Section 7.2.1] and the rest of this data structure can be constructed in $O(g) \subseteq O(n)$ time.

4.4 Even Faster Queries

We provide a faster solution in the case where pattern P is not chosen *adversarially*. With this term, we mean that substrings of P do not collide with substrings of T through the hash functions used by the structures of Lemmas 25 and 26 with high probability; in other words, we assume that an adversary did not craft P so that it generates such collisions. This is a realistic scenario in practice since a string chosen independently from (the random choices in) those data structures will collide with a text substring of T through those hash functions with low (inverse polynomial) probability only (see [40, 41]):

Remark 29 ([40, 41]) The structures of Lemma 25 and Lemma 26 return the correct result with high probability, assuming that their inputs (respectively, $P[i, j]$ and β) are not chosen by an adversary on the basis of the random seeds in the hash functions used by those data structures. To abbreviate, in such a case we will say that the inputs are " not chosen adversarially ".

Observe that the non-adversarial hypothesis is the same required in other randomized data structures such as classic hash tables and Bloom filters [43] (where the random choices precede the arrival of the input) and is common in other works using z-fast tries to index text [40].

Algorithm 2 can be understood as a traversal of $d \leq m$ suffix tree edges, where d is the number of times we would fully or partially descend edges in the suffix tree of T while finding MEMs (a generalization to the context of MEMs of parameter d used in Theorem 18). Figure 2 shows an example where d is much smaller than m.

By following the same analysis carried out in the proof of Theorem 27 but using the data structure in Lemma 25 for computing $LCP()$, we immediately obtain:

Theorem 30 *Let $T[1,n]$ be a text with Suffixient Array size χ for which there exists a SLP of size g. Then Algorithm 2, when implemented with the structure of Lemma 26 to support operation $search()$ and with the structure of Lemma 25 to support function $LCP()$, finds all MEMs of any pattern $P[1,m]$ in T in $O(m + d \log n)$ time using $O(\chi + g)$ words of space, where $d \leq m$ is the number of times we would fully or partially descend edges in the suffix tree of T while finding those MEMs. The result is correct with high probability, provided that P is not chosen adversarially.*

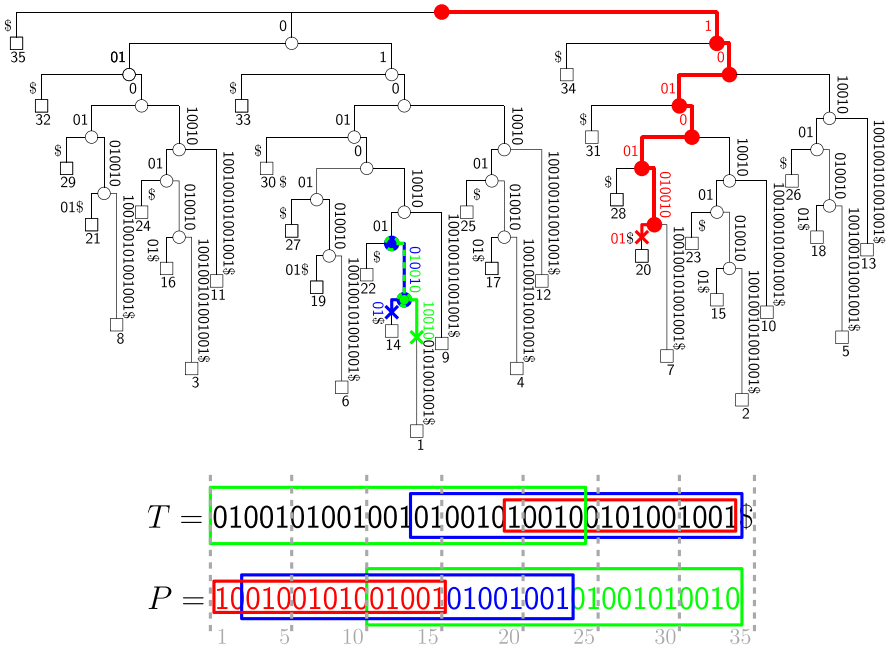

Fig. 2 The $d = 11$ times we fully or partially descend 10 distinct edges in the suffix tree of T **(above)** while finding the 3 MEMs for our example **(below)**. The MEMs are shown boxed in P and T, with the characters' colors in P also indicating which path we are following in the tree when we read them. The characters in the box for a MEM that are a different color from the box are the path label of the node we reach by suffix links and descend from when finding the end of that MEM. We descend the line alternating blue and green twice. Notice that $11 = d \ll m = 34$

4.5 Offline Solutions with only Random Access and z-Fast Tries

Interestingly, fast random access to T and a z-fast trie on the set of strings $Z = \{T[1, j] \ j \in \mathrm{sA}\}$ are by themselves sufficient for fast offline pattern matching. To see why, assume P does occur in T and we already know the location of an occurrence $T[j - i + 1, j]$ of $P[1, i]$ in T. If $T[j + 1] = P[i + 1]$ then $T[j - i + 1, j + 1] = P[1, i + 1]$. Otherwise, querying the z-fast trie with $P[1, i + 1]$ will return in $O(\log(i + 1)) \subseteq O(\log m)$ time a position $j' + 1$ such that $T[j' - i + 1, j' + 1] = P[1, i + 1]$.

It follows that in $O(t_a \cdot m + d \log m)$ time we can find a position j^* such that if P occurs in T then $T[j^* - m + 1, j^*] = P$, which we can then check in $O(t_a \cdot m)$ time. Here, $d \leq m$ is again the number of times we would fully or partially descend edges in the suffix tree of T while finding the MEMs of P with respect to T (so the node depth of the locus of P if P occurs in T). If P occurs in T then our time bound holds in the worst case; otherwise, it holds with high probability assuming an adversary has not chosen P on the basis of the hash function used in our z-fast trie. In any case, our solution works in $O(m(t_a + \log m))$ time.

This solution is offline in the sense that we may not know until we have finished whether any prefix $P[1, i]$ of P occurs in T. Moreover, if P does not occur in T then we do not learn which is the longest prefix of P that occurs in T.

Theorem 31 *Suppose we already have t_a-time random access to a text $T[1, n]$ with Suffixient Array size χ. Then we can store in $O(\chi)$ space a z-fast trie such that in $O(m(t_a + \log m))$ time we can check whether $P[1, m]$ occurs in T and, if so, return the location of one occurrence.*

We can adapt this idea to offline MEM-finding. Suppose T has a Suffixient Array sA of size χ, T^{rev} has a Suffixient Array sA$'$ of size χ', and we have fast random access to T (and thus also to T^{rev}) and z-fast tries for the sets of strings $Z = \{T[1, j] \ j \in \mathrm{sA}\}$ and $Z' = \{T^{\mathrm{rev}}[1, j] \ j \in \mathrm{sA}'\}$. These z-fast tries take $O(\chi + \chi')$ total space.

Assume we already know the ending position j of a prefix $T[1, j]$ of T with the longest common suffix with $P[1, i]$ of any prefix of T, possibly without knowing the length of that common suffix. If $T[j + 1] = P[i + 1]$ then $T[1, j + 1]$ has the longest common suffix with $P[1, i + 1]$ of any prefix of T. Otherwise, querying the z-fast trie for Z with $P[1, i + 1]$ will return in $O(\log(i + 1)) \subseteq O(\log m)$ time a position $j' + 1$ such that $T[1, j' + 1]$ has the longest common suffix with $P[1, i + 1]$ of any prefix of T.

It follows that in $O(t_a \cdot m + d \log m))$ time we can build an array $J[1, m]$ such that $T[1, J[i]]$ has the longest common suffix with $P[1, i]$ of any prefix of T, for $i \leq m$, where $d \leq m$ is again the number of times we would fully or partially descend edges in the suffix tree of T while finding the MEMs of P with respect to T.

Because the prefixes of T^{rev} are the suffixes of T, in $O(t_a \cdot m + d' \log m)$ time we can also build an array $J'[1, m]$ such that $T[J'[i], n]$ has the longest common prefix with $P[i, m]$ of any suffix of T, where d' is the number of times we would fully or partially descend edges in the suffix tree of T^{rev} while finding the MEMs of P^{rev} with respect to T^{rev}.

We can find the ending position e_1 of the leftmost MEM $P[1, e_1]$ in P in $O(t_a \cdot e_1)$ time, by comparing the characters in $P[1, m]$ and $T[J'[1], n]$ until we find a mismatch at $P[e_1 + 1]$. Since $P[e_1 + 1]$ is in the second MEM $P[s_2, e_2]$ in left-to-right order, we can find that MEM's starting position s_2 in $O(t_a(e_1 - s_2 + 1))$ time by comparing the characters in $P[1, e_1 + 1]$ and $T[1, J[e_1 + 1]]$ from right to left until we find a mismatch at $P[s_2 - 1]$.

Since the second MEM in P is the first MEM in $P[s_2, m]$, we can repeat this whole process to find the ending position of the second MEM and the starting position of the third MEM, and so on. This step is similar to Li's [44] forward-backward algorithm and takes time proportional to t_a times the total length of the MEMs (we could also use as a base an algorithm from [45] and [34, Section 2.6], which is faster in practice).

With high probability, our algorithm works correctly and in $O(t_a \cdot m + (d + d') \log m)$ time plus time proportional to t_a times the total length of the MEMs, assuming an adversary has not chosen P on the basis of the hash function used in our z-fast tries.

Theorem 32 *Suppose we already have t_a-time random access to a text $T[1, n]$ with Suffixient Array size χ whose reverse has Suffixient Array size χ'. Then we can store in $O(\chi + \chi')$ space two z-fast tries such such that with high probability we can find the MEMs of $P[1, m]$ with respect to T in $O(t_a \cdot m + (d + d') \log m))$ time plus time proportional to t_a times the total length of the MEMs, assuming an adversary has not chosen P on the basis of the hash function used in our z-fast trie.*

5 Characterization of Smallest Suffixient Sets

We now move to the problems of computing Suffixient Arrays. We start in this section by characterizing smallest suffixient sets. We first define the concept of *supermaximal extensions* (Definition 33). Then, in Definition 34 we define a set (actually, a family of sets) $\mathcal{S}$ containing all positions in T "capturing" all (and only the) supermaximal extensions of T. Finally, in Lemma 35 we prove that $\mathcal{S}$ is a suffixient set of minimum cardinality.

Definition 33 We say that $T[i, j]$ (with $j \geq i$) is a *supermaximal extension* if $T[i, j - 1]$ is right-maximal and, for each right-maximal $T[i', j' - 1] \neq T[i, j - 1]$ (with $i' \leq j' \leq n$), $T[i, j]$ is not a suffix of $T[i', j']$.

The following definition depends on a particular total order $<_t$ on $[n]$. We use $<_t$ to break ties between equivalent candidate positions of the suffixient set, choosing the position of maximum rank according to $<_t$ in case of a tie. In order to make notation lighter, we do not parameterize the set on the particular tie-breaking strategy ($<_t$ will always be clear from the context).

Definition 34 Let $<_t$ be any total order on $[n]$. We define a set $\mathcal{S} \subseteq [n]$ as follows: $x \in \mathcal{S}$ if and only if there exists a supermaximal extension $T[i, j]$ such that (i) $T[i, j]$ is a suffix of $T[1, x]$, and (ii) for all prefixes $T[1, y]$ suffixed by $T[i, j]$, if $y \neq x$ then $y <_t x$.

We prove that $\mathcal{S}$ is a suffixient set of minimum cardinality:

Lemma 35 *For any tie-breaking strategy (total order) $<_t$, $\mathcal{S}$ is a suffixient set of minimum cardinality χ for T.*

Proof To see that $\mathcal{S}$ is suffixient, consider any right-maximal substring $T[i, j-1]$ ($i \leq j \leq n$). We want to prove that there exists $x \in \mathcal{S}$ such that $T[i, j]$ is a suffix of $T[1, x]$. Let $T[1, x]$ be a prefix of T being suffixed by $T[i, j]$, breaking ties (on its endpoint x) by $<_t$. If $T[i, j]$ is a supermaximal extension, then by Definition 34 it holds $x \in \mathcal{S}$ and we are done. Otherwise, let $T[i', j'] \neq T[i, j]$ be a one-character extension of a right-maximal string $T[i', j'-1]$ such that $T[i, j]$ is a suffix of $T[i', j']$.

Without loss of generality, let $T[i', j']$ be the string of maximum length $|T[i', j']| = j' - i' + 1$ with this property. Observe that there cannot be a right-extension $T[i'', j''] \neq T[i', j']$ of a right-maximal string $T[i'', j''-1]$ such that $T[i', j']$ is a suffix of $T[i'', j'']$, otherwise $T[i, j]$ would be a suffix of $T[i'', j'']$ and $|T[i'', j'']| > |T[i', j']|$, contradicting the fact that $T[i', j']$ is the longest such string. It follows that $T[i', j']$ is a supermaximal extension. Let $T[1, y]$ be a prefix of T being suffixed by $T[i', j']$, breaking ties (on y) by $<_t$. Then, by Definition 34, $y \in \mathcal{S}$. Also in this case we are done, since $T[i, j]$ suffixes $T[i', j']$ and $T[i', j']$ suffixes $T[1, y]$, so by transitivity of the suffix relation, $T[i, j]$ suffixes $T[1, y]$.

To see that $\mathcal{S}$ is of minimum cardinality, let $\mathcal{S}'$ be a suffixient set. We prove $|\mathcal{S}| \leq |\mathcal{S}'|$ by exhibiting an injective function from $\mathcal{S}$ to $\mathcal{S}'$.

Let $SE \subseteq \Sigma^+$ be the set of all supermaximal extensions. We preliminarily show that any prefix $T[1, j]$ can be suffixed by at most *one* supermaximal extension: assume, for a contradiction, that $T[1, j]$ is suffixed by two distinct supermaximal extensions $T[i, j] \neq T[i', j]$. Without loss of generality, $i' < i$. But then, $T[i, j]$ is a suffix of $T[i', j]$, hence $T[i, j]$ cannot be a supermaximal extension, a contradiction.

We first define a relation $h : \mathcal{S} \to SE$ mapping every $x \in \mathcal{S}$ to a supermaximal extension $h(x) = T[i, j]$ such that $T[i, j]$ is a suffix of $T[1, x]$. From the observation above, there is at most one such string $T[i, j]$. From Definition 34, there exists one such $T[i, j]$. We conclude that h is a function. We now prove that h is injective.

To see that h is injective, assume for a contradiction $h(x) = h(y)$ for some $x \neq y$ with $x, y \in \mathcal{S}$. Let $h(x) = h(y) = T[i, j]$.

By Definition 34, there exists a supermaximal extension $T[i_x, x]$ such that $T[1, x]$ is the prefix of T being suffixed by $T[i_x, x]$ with largest endpoint x, according to the total order $<_t$. Following the same reasoning on position y, we can associate an analogous supermaximal extension $T[i_y, y]$ to y. On the other hand, by the definition of h, also string $h(x) = h(y) = T[i, j]$ is a supermaximal extension that suffixes both $T[1, x]$ and $T[1, y]$. Since $T[i, j]$ and $T[i_x, x]$ are supermaximal extensions that suffix $T[1, x]$ and (as proved above) there can exist at most one supermaximal extensions suffixing $T[1, x]$, we conclude that $T[i, j] = T[i_x, x]$. Similarly, we conclude $T[i, j] = T[i_y, y]$, hence $T[i_x, x] = T[i_y, y]$. We obtained a contradiction, since Definition 34 requires both $x <_t y$ and $y <_t x$ and $<_t$ is a total order. We conclude that h is an injective function.

Next, we define a relation $g : SE \to \mathcal{S}'$ from the set SE of supermaximal extensions to positions in $\mathcal{S}'$. For any supermaximal extension $T[i, j] \in SE$, we define $g(T[i, j]) = x$ to be the largest element of $\mathcal{S}'$ (according to the standard order between integers) such that $T[1, x]$ is suffixed by $T[i, j]$. Such a position must exist since $\mathcal{S}'$

is a suffixient set, so g is a function. To see that g is also injective, assume for a contradiction $g(T[i, j]) = g(T[i', j']) = x$, for $T[i, j] \neq T[i', j']$ (both supermaximal extensions). By the definition of g, both $T[i, j]$ and $T[i', j']$ are suffixes of $T[1, x]$. But then, since $T[i, j] \neq T[i', j']$, either $T[i, j]$ is a suffix of $T[i', j']$ (so $T[i, j]$ is not a supermaximal extension) or the other way round (so $T[i', j']$ is not a supermaximal extension). In both cases we get a contradiction, so we conclude that g is injective.

We conclude the proof by observing that $g \circ h : \mathcal{S} \to \mathcal{S}'$ is the composition of two injective functions and is therefore injective. □

6 Computing Suffixient Arrays

To make notation lighter, in this section we focus on the core problem of computing smallest suffixient sets, i.e., (unsorted) sets $\{x \;\; x \in \mathrm{sA}\}$. By their nature, it will be immediate to augment our algorithms so that they output also the Prefix Array rank of every $x \in \mathrm{sA}$; the Suffixient Array sA can then be obtained by sorting the positions $x \in \mathrm{sA}$ according to those ranks.

We start in Section 6.1 by giving an "operative" version of Definition 34. This "operative" definition will automatically yield a simple quadratic-time algorithm for computing a smallest suffixient set. In the next subsections, we will optimize this algorithm and reach compressed working space (Section 6.2) and optimal linear time (Sections 6.3 and 6.4).

6.1 A Simple Quadratic-Time Algorithm

For brevity, in the rest of the paper, let LCP, BWT, SA denote $\mathrm{LCP}(T^{\mathrm{rev}})$, $\mathrm{BWT}(T^{\mathrm{rev}})$, and $\mathrm{SA}(T^{\mathrm{rev}})$, respectively.

First, the starting and ending positions of runs on BWT are of our particular interest:

Definition 36 (c-run break) We say that position $1 < i \leq n$ is a *c-run break*, for $c \in \Sigma$, if $\mathrm{BWT}[i-1, i] = ac$ or $\mathrm{BWT}[i-1, i] = ca$, with $a \in \Sigma$ and $a \neq c$.

Note that, if i is a c-run break, then it is also an a-run break for some $a \neq c$.

Definition 37 Given a position $i \in [n]$, we define $box(i) = [\ell, r]$ to be the maximal interval such that $i \in [\ell, r]$ and $\mathrm{LCP}[j] \geq \mathrm{LCP}[i]$ for all $j \in [\ell, r]$.

Example 38 See Fig. 3: $box(10) = [3, 13]$ (shown with a blue box), because $\mathrm{LCP}[3, 13] \geq \mathrm{LCP}[10] = 1$, and $box(19) = [17, 20]$ (shown with an orange box).

Next, we define the set of c-run breaks in $box(i)$.

Definition 39 Given $i \in [n]$ and $c \in \Sigma$, we define

$$B_{i,c} = \{j \;:\; j \text{ is a } c\text{-run break in } \mathrm{BWT}(T^{\mathrm{rev}}) \text{ and } j \in box(i)\}.$$

Example 40 In Fig. 3, we have $B_{10,A} = \{3, 6, 7, 10, 11, 12\}$.

We indicate with $text(i) = n - \mathrm{SA}[i] + 1$ the position in T corresponding to character $\mathrm{BWT}[i]$, and with $bwt(j) = \mathrm{SA}^{-1}[n + 1 - j]$ the inverse of function $text()$, i.e., the position in BWT corresponding to character $T[j]$.

The following Lemma 41 stands at the core of all our algorithms for computing a smallest suffixient set. Intuitively, Lemma 41 states that supermaximal extensions can be identified by looking at local maxima among c-run breaks contained in LCP ranges of the form $\mathrm{LCP}[box(i)]$. Since, by Definition 34, supermaximal extensions are those characterizing suffixient sets of smallest cardinality, Lemma 41 gives us a tool for computing such a set using the arrays BWT, LCP, and SA. See Fig. 3.

Lemma 41 *The following hold:*

1. *Let i be a c-run break and $i' \in \{i, i-1\}$ be such that $\mathrm{BWT}[i'] = c$. Let $j' = text(i')$, and let $\ell = \mathrm{LCP}[i]$. If $\ell = \max \mathrm{LCP}[B_{i,c}]$, then the string $T[j'-\ell, j']$ is a supermaximal extension, and*
2. *conversely, for any supermaximal extension $\alpha \cdot c$ ($\alpha \in \Sigma^*$, $c \in \Sigma$) there exists an occurrence $\alpha \cdot c = T[j'-\ell, j']$ and a c-run break $i \in [n]$ such that $\mathrm{BWT}[i'] = c$, with $i' = bwt(j') \in \{i, i-1\}$ and $\ell = \mathrm{LCP}[i] = \max \mathrm{LCP}[B_{i,c}]$.*

Proof *(1)* Let i be a c-run break. Let $i', i'' \in \{i, i-1\}$ be such that $\mathrm{BWT}[i'] = c$ and $\mathrm{BWT}[i''] = a \neq c$. Let $j' = text(i')$ and $j'' = text(i'')$. Let moreover $\ell = \mathrm{LCP}[i]$, and assume $\ell = \max \mathrm{LCP}[B_{i,c}]$.

Note that $T[j'-\ell, j'-1]$ is right-maximal: $\ell = \mathrm{LCP}[i]$ implies that $T[j'-\ell, j'-1] = T[j''-\ell, j''-1]$ and, by definition of i', i'', j', and j'' we have that $T[j'] = \mathrm{BWT}[i'] \neq \mathrm{BWT}[i''] = T[j'']$.

Assume, for a contradiction, that $T[j'-\ell, j']$ is not a supermaximal extension. Then, since $T[j'-\ell, j'-1]$ is right-maximal, it must be the case that $T[j'-\ell, j']$ is a suffix of some string $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}]$ with $\hat{\ell} > \ell$ and such that $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}-1]$ is right-maximal. Let $\hat{\imath} = bwt(\hat{\jmath})$. Since $T[j'-\ell, j']$ is a suffix of $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}]$ we have that $T[j'-\ell, j'-1]$ is a suffix of $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}-1]$, hence it must be the case that $\hat{\imath} \in box(i)$. Since $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}-1]$ is right-maximal and $T[\hat{\jmath}] = c$, without loss of generality we can assume that $\hat{\imath} \in \{k-1, k\}$, where k is a c-run break (such a c-run break must exist in $box(i)$) and $\mathrm{LCP}[k] \geq \hat{\ell}$. In particular, $k \in box(i)$. Then, $\mathrm{LCP}[k] \geq \hat{\ell} > \ell$ and $k \in box(i)$ yield $\max \mathrm{LCP}[B_{i,c}] \geq \hat{\ell} > \ell$, a contradiction.

(2) Let $T[j'-\ell, j']$ be a supermaximal extension and let $c = T[j']$. Let $i' = bwt(j')$. Since $T[j'-\ell, j'-1]$ is right-maximal, we can assume without loss of generality that either $\mathrm{BWT}[i'] \neq \mathrm{BWT}[i'-1]$ or $\mathrm{BWT}[i'] \neq \mathrm{BWT}[i'+1]$. Let therefore $i' \in \{i, i-1\}$, where i is a c-run break. By definition of $T[j'-\ell, j']$, it holds $\ell = \mathrm{LCP}[i]$ (because $T[j'-\ell, j'-1]$ is the longest right-maximal string suffixing $T[1, j'-1]$). We need to prove that $\ell = \max \mathrm{LCP}[B_{i,c}]$. Assume, for a contradiction, that $\ell < \hat{\ell} = \max \mathrm{LCP}[B_{i,c}]$. This means that there exists a supermaximal extension $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}]$ with (i) $T[\hat{\jmath}] = c$, (ii) $\hat{\imath} \in \{k-1, k\}$, where $\hat{\imath} = bwt(\hat{\jmath})$ and $k \in B_{i,c}$ is a c-run break in $box(i)$, and (iii) $\mathrm{LCP}[k] = \hat{\ell}$. But then, since $k \in box(i)$ and $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}-1]$ is a right-maximal string of length $\hat{\ell} > \ell$, we have that $T[j'-\ell, j'-1]$ is a (proper) suffix of $T[\hat{\jmath}-\hat{\ell}, \hat{\jmath}-1]$. This fact and $T[\hat{\jmath}] = c$ contradict the fact that $T[j'-\ell, j']$ is a supermaximal extension. □

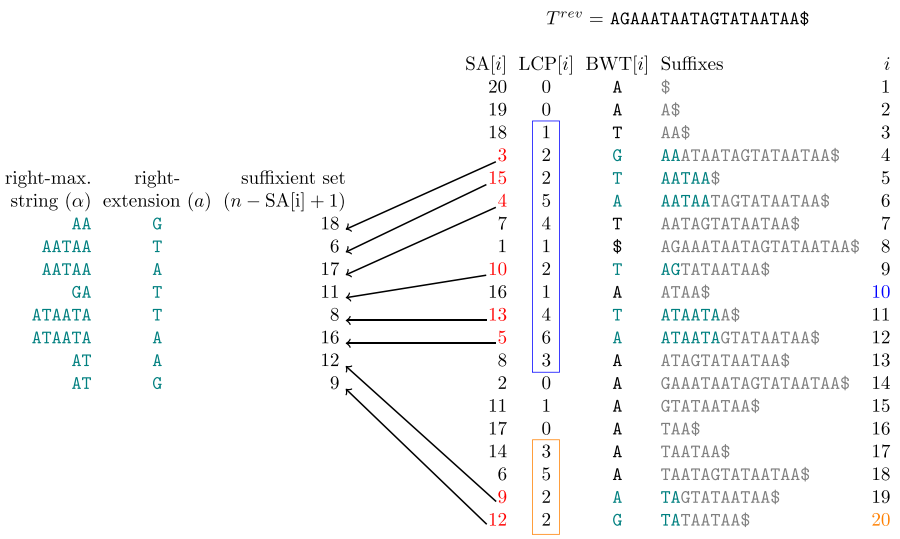

Fig. 3 The figure shows how to construct a smallest suffixient set $\mathcal{S}$ for a text $T[1, n]$ with $n = 20$ following Lemma 41. *To the left of the black arrows*: all supermaximal extensions $\alpha \cdot a$ of T and their selected ending positions in T, forming a suffixient set. *To the right of the black arrows*: SA, LCP, BWT and the sorted suffixes of T^{rev}. In column SA[i], we highlight in red all positions that are selected to be included in $\mathcal{S}$. In columns *Suffixes* and BWT[i], we highlight in green the (reverses of) the supermaximal extensions of T. Black arrows show how the selected SA positions are converted to positions in T using the formula $n - \text{SA}[i] + 1$. *How to identify positions of* $\mathcal{S}$: for each c-run break i, we decide if the two ranks $i' \in \{i - 1, i\}$ should contribute to $\mathcal{S}$ (i.e., if they correspond to a supermaximal extension) as described in Lemma 41. For brevity, we show this decisional procedure only on two run breaks. Consider the A-run break at position $i = 10$ (highlighted in blue in column i). The blue box depicts the corresponding LCP interval, $\text{LCP}[box(10)] = \text{LCP}[3, 13]$. We observe that in $[3, 13]$ there are other A-run breaks i'', such that $\text{LCP}[i''] > \text{LCP}[10]$: those are $i'' = 6, 7, 11, 12$. We conclude that text position $n - \text{SA}[10] + 1 = 5$ should not be included in $\mathcal{S}$. Consider now the A-run break $i = 20$, highlighted in orange in column i. Position $i' = 20 - 1 = 19$ is such that $\text{BWT}[i'] = \text{A}$. The orange box depicts the corresponding LCP interval, $\text{LCP}[box(20)] = \text{LCP}[17, 20]$. In this case, there is no other A-run break in $[17, 20]$ with an LCP value larger than $\text{LCP}[20] = 2$. We therefore insert text position $n - SA[i'] + 1 = n - SA[19] + 1 = 12$ in $\mathcal{S}$. Notice that $i = 20$ is also a G-run break; repeating the above reasoning, one can verify that position $i' = 20$ is indeed associated with the supermaximal extension $\text{AT} \cdot \text{G}$ ending in text position $n - SA[20] + 1 = 9$

We immediately obtain a simple quadratic algorithm computing a suffixient set of smallest cardinality: see Algorithm 3. Note that, in Lemma 41, if $i'' \in B_{i,c}$ and $\text{LCP}[i''] = \text{LCP}[i]$, then $box(i) = box(i'')$. Since there could be multiple c-run breaks $i'' \in B_{i,c}$ such that $\text{LCP}[i''] = \text{LCP}[i]$, if $\text{LCP}[i] = \text{LCP}[i''] = \max \text{LCP}[B_{i,c}]$ then we have to break ties and insert only one of the corresponding text positions in the suffixient set. In Line 7 of Algorithm 3 we choose the largest such c-run break: as a matter of fact, this defines a particular tie-breaking strategy $<_t$ between positions $[n]$ (as required by Definition 34); i.e., for $y, x \in [n]$ $y <_t x$ iff $T[1, y] <_{\text{colex}} T[1, x]$.

Computing BWT, LCP, and SA in Line 1 takes $O(n)$ time. Additionally, for each $i = 2, \ldots, n$, Algorithm 3 computes the set $B_{i,c}$ in line 7 by scanning $\text{LCP}[\ldots, i - 1, i, i + 1, \ldots]$ in order to identify $\text{LCP}[box(i)]$ ($O(n)$ time for each such scan). It follows that the total running time is bounded by $O(n^2)$. Correctness follows immediately from Lemma 41, Definition 34, and Lemma 35.

Algorithm 3 Quadratic algorithm computing a smallest suffixient set.

```
   input : A text T[1, n] ∈ Σ^n
   output: A suffixient set S of smallest cardinality for T.
 1 BWT ← BWT(T^rev); LCP ← LCP(T^rev); SA ← SA(T^rev);
 2 S ← ∅;
 3 for i = 2, ..., n do
 4     if BWT[i − 1] ≠ BWT[i] then
 5         for i' ∈ {i − 1, i} do
 6             c ← BWT[i'] ;                          /* i is a c-run break */
 7             if i = max{i'' : i'' ∈ B_{i,c} ∧ LCP[i''] = max LCP[B_{i,c}]} then
 8                 S ← S ∪ {n − SA[i'] + 1}
 9             end
10         end
11     end
12 end
13 return S;
```

6.2 A One-Pass Algorithm

In this section, we speed up the simple quadratic algorithm provided in the previous section. We will refer to this version as "one-pass" since it only requires one scan of the BWT, SA, and LCP arrays for T^{rev}. The algorithm is summarized in Algorithms 4 and 5. See below for a description.

Algorithm 4 One-pass algorithm building a smallest suffixient set.

```
   input : A text T[1, n] over a finite alphabet Σ.
   output: A suffixient set S of smallest cardinality for T.
 1 S ← ∅ ;                          /* initialize the empty suffixient set S */
 2 BWT ← BWT(T^rev); LCP ← LCP(T^rev); SA ← SA(T^rev);
 3 R[1, σ] ← ((−1, 0, false), ..., (−1, 0, false)) ;          /* LCP local maxima */
 4 m ← ∞;                          /* LCP minimum inside current BWT run */
 5 for i = 2, ..., n do
 6     m ← min{m, LCP[i]};
 7     if BWT[i] ≠ BWT[i − 1] then
 8         eval(Σ, m, R, S);
 9         for i' ∈ {i − 1, i} do
10             if LCP[i] > R[BWT[i']].len then
11                 R[BWT[i']] ← (LCP[i], n − SA[i'] + 1, true);
12             end
13         end
14         m ← ∞;
15     end
16 end
17 eval(Σ, −1, R, S) ;                /* evaluate last active candidates */
18 return S;
```

Observe that Algorithm 3 runs in quadratic time due to the fact that it needs to scan LCP[$box(i)$] for each run break i, and those boxes overlap. The main intuition of Algorithm 4 is that instead of scanning $box()$ for every position, we simply detect the

Algorithm 5 Procedure $eval(C, l, R, \mathcal{S})$.

```
input : A set of characters C, a LCP value l, the candidate's list R and the suffixient set S.
output: The updated suffixient set S.
foreach c ∈ C do
    if l < R[c].len then
        if R[c].active then
            S ← S ∪ {R[c].pos};
        end
        R[c] ← {l, 0, false};
    end
end
```

end of the corresponding box for each suffixient set position. Suppose we already know a position $j \in [n]$ such that j corresponds to a suffixient set position $n - SA[j] + 1$ to be reported. Note that by definition, LCP$[j]$ is the maximum LCP value in $B_{j,c}$ where $c \in \Sigma$ is such that j is a c-break. Let ℓ, r be such that $box(j) = [\ell, r]$. Then by definition, it must hold that $r = n$ or LCP$[r+1] <$ LCP$[j]$. Therefore, if we keep track of such a position j with a locally maximal LCP value for each $c \in \Sigma$, we can determine whether its corresponding box $box(j)$ ends at a particular position by simply comparing the LCP value. Note that it is not necessary to check this at every position. Instead, it is sufficient to check at the end of every BWT-run whether the corresponding box has already ended, by performing the comparison with the minimum LCP value within each BWT-run.

Let us describe Algorithm 4 in detail. Assume we scanned the arrays BWT, SA, and LCP up to position i. R is a map associating each $c \in \Sigma$ with information related with some c-run break $j \leq i$ corresponding to a candidate supermaximal extension $\alpha \cdot c$. Each entry $R[c] = (len, pos, active)$ is composed of three values: $R[c].len =$ LCP$[j]$ (i.e., the length of the right-maximal string α), $R[c].pos = text(j') = n -$ SA$[j'] + 1$, where $j' \in \{j-1, j\}$ is such that BWT$[j'] = c$ (i.e., the position of the last character of the supermaximal extension $\alpha \cdot c$ in T), and a boolean flag $R[c].active$ which is set to *true* if and only if $R[c].len$ is the maximum value in LCP$[B_{j,c} \cap [i]]$ (breaking ties by smaller j: we keep the first encountered such maximum). Depending on the value of flag $active$, we will distinguish between *active* and *inactive candidates*.

Example 42 See Fig. 3 and assume we scanned BWT up to position $i = 4$. The candidate associated with letter T is $R[\mathtt{T}] = (2, 3, true)$, where $R[\mathtt{T}].len =$ LCP$[4] =$ 2 and $R[\mathtt{T}].pos = n - SA[3] + 1 = 3$. This candidate is active since LCP$[4]$ is the largest value in LCP$[B_{4,\mathtt{T}} \cap [4]] = \{2\}$. When processing position $i = 6$, we find another T-run break. Here we update $R[\mathtt{T}] =$ (LCP$[i], n - SA[i'] + 1, true) =$ $(5, 6, true)$ where $i' = i - 1 = 5$ since $6 \in B_{4,\mathtt{T}} \cap [6]$, and $5 =$ LCP$[6] >$ LCP$[4] = 2$.

Algorithm 4 works as follows. We scan the BWT left-to-right: BWT[2], ..., BWT$[i]$ (for $i = 2, \ldots, n$). For $c \in \Sigma$, let $j < i$ be the c-run break whose information is stored in $R[c]$, i.e., such that LCP$[j] = R[c].len$. If LCP$[i] < R[c].len$, then $i \notin box(j)$. But then, if $R[c].active = true$, position $R[c].pos$ must belong to the output set so we have to insert it in $\mathcal{S}$. We also set $R[c].active = false$ in order to

record that the maximum in LCP[$B_{j,c}$] has been found (if $R[c].active$ is already equal to *false*, $\mathcal{S}$ does not need to be updated since the maximum in LCP[$B_{j,c}$] had already been found). Observe that these operations, performed by the call to $eval(\Sigma, m, R, \mathcal{S})$ (lines 8,17) (see below for the meaning of variable m), have to be repeated for each $c \in \Sigma$; however, since i can possibly replace the candidate in $R[c]$ only if i is a c-run break, we can avoid calling $eval(\cdot)$ inside equal-letter BWT runs as follows: we compute the minimum LCP value m inside LCP intervals corresponding to BWT equal-letter runs (line 6), and perform the check $m < R[c].len$ for each $c \in \Sigma$ by calling procedure $eval(\Sigma, m, R, \mathcal{S})$ (costing time $O(\sigma)$) only when the current position i is a run break. This ultimately reduces the algorithm's running time from $O(n \cdot \sigma)$ to $O(n + \bar{r} \cdot \sigma)$.

On the other hand, if i is a c-run break and LCP[i] $> R[c].len$, then $i \in B_{j,c} \cap [i]$ and $R[c].len$ is not a local maximum in LCP[$B_{j,c}$] (line 10). In this case, we have to replace the candidate stored in $R[c]$ with the information associated with the new candidate i: letting $i' \in \{i-1, i\}$ be such that BWT[i'] $= c$, we replace $R[c] \leftarrow$ (LCP[i], $n - $ SA[i'] $+ 1$, *true*) (line 11).

Example 43 Continuing Example 42. When processing the T-run break $i = 7$, we see that LCP[7] $< R[\mathtt{T}].len$. Thus, we reached the end of $box(6) = [6, 6]$, where max LCP[$B_{6,\mathtt{T}}$] $= R[\mathtt{T}].len = 5$, and insert $R[\mathtt{T}].pos$ in $\mathcal{S}$. Next, we update $R[\mathtt{T}] = (4, 0, false)$. On $i = 8$, we again update $R[\mathtt{T}] = (1, 0, false)$. Finally, on $i = 9$, we get LCP[9] $> R[\mathtt{T}].len$ since $box(6)$ and $box(9)$ are now disjoint. We update $R[\mathtt{T}] =$ (LCP[9], 11, *true*) to the active state.

We obtain:

Lemma 44 *OnepassProof Given a text $T[1, n]$ over alphabet of size σ, Algorithm 4 computes a smallest suffixient set $\mathcal{S}$ in $O(n + \bar{r} \cdot \sigma)$ time and $O(n)$ words of space.*

Proof Algorithm 4 scans the BWT, LCP, and SA exactly once and, for each run break, checks if any of the σ candidates is a supermaximal extension using Algorithm 5. Since Algorithm 5 performs a constant number of operations for each $c \in \Sigma$, the whole algorithm runs in $O(n + \bar{r} \cdot \sigma)$ time. In addition, the only supplementary data structure we need is R, which takes $O(\sigma)$ words of space; thus if $\sigma < n$, altogether, we take $O(n)$ words of space.

We prove the correctness of Algorithm 4 by showing it computes the same output of Algorithm 3. In particular, given $\mathcal{S}'$, the output of Algorithm 4, and $\mathcal{S}$, the output of Algorithm 3, we show that given any value $s = n - $ SA[i] $+ 1$, where $i \in [n]$, *(1)* $s \in \mathcal{S} \implies s \in \mathcal{S}'$ and *(2)* $s \notin \mathcal{S} \implies s \notin \mathcal{S}'$.

(1) Let $(n - $ SA[i'] $+ 1) \in \mathcal{S}$, where i is a c-run break, such that $i' \in \{i-1, i\}$ and BWT[i'] $= c$. By Lemma 41, it follows that max LCP[$B_{i,c}$] $=$ LCP[i]. Assume we scanned the BWT up to position i, there are three cases: i' is the position of the first occurrence of c in the BWT, or there exists another c-run break at position $j < i$, such that either $i \in B_{j,c}$ and LCP[i] $>$ LCP[j], or $box(i) \cap box(j) = \emptyset$. For all these three cases, LCP[i] $> R[c].len$; thus, i is set as the new active c candidate in R (lines 10-12). Now, if i' is the position of the last occurrence of c in the BWT, $R[c].pos$ is inserted in $\mathcal{S}$ at the end of the algorithm (line 17); otherwise,

let $box(i) = [b, e]$, where $b \leq i \leq e$, we have $\forall s \in \text{LCP}[b, e]$, $s \geq \text{LCP}[i]$ and no $j' \in [i+1, b]$ is a c-run break such that $\text{LCP}[j'] > \text{LCP}[i]$. Thus, $R[c]$ is not updated until we scan $\text{LCP}[e+1]$. At this point, $R[c].len > \text{LCP}[e+1]$ and since $R[c]$ is an active candidate; thus, Algorithm 4 insert $R[c].pos = (n - \text{SA}[i] + 1)$ in $\mathcal{S}'$ (Algorithm 5, line 4).

(2) Now let $(n - \text{SA}[i'] + 1) \notin \mathcal{S}$, it follows by Lemma 41, that $\exists j \in B_{i,c}$ such that $\text{LCP}[j] > \text{LCP}[i]$ (for simplicity we consider the smallest j). Assume we scanned the BWT up to position i; we need to consider two cases: *(i)* $j > i$ and *(ii)* $j < i$. *(i)* If $j > i$, let $R[c].len = \text{LCP}[i]$, then $\forall l \in \text{LCP}[i+1, j]$, $l \geq R[c].len$; thus, $(n - \text{SA}[i'] + 1)$ is not inserted in $\mathcal{S}$ (lines 2-6, Algorithm 5) until position j. Here, we get $\text{LCP}[j] > R[c].len$ and, update $R[c].pos = (n - \text{SA}[j'] + 1)$ (lines 10-12), where $j' \in \{j-1, j\}$ and $\text{BWT}[j'] = c$. Due to this, $(n - \text{SA}[i'] + 1)$ is dropped and not inserted in $\mathcal{S}'$. *(ii)* If $j < i$, then $\exists l \in \text{LCP}[j+1, i]$, $l < \text{LCP}[j]$; thus when we read l, $R[c]$ is updated to the inactive state (line 6, Algorithm 5). However, since $\forall s \in B_{i,c} \cap [j+1, i]$, $\text{LCP}[s] < \text{LCP}[j]$ it means that $R[c]$ is never updated to the active state for any position in $[j+1, i]$ (we skip lines 10-12), so also in this case $(n - \text{SA}[i'] + 1)$ is not inserted in $\mathcal{S}'$. □

Observe that in Algorithm 5 (procedure $eval(\cdot)$), only entries such that $l < R[c].len$ are possibly modified. This suggests that R could be sorted in order to speed up operations. Indeed, it turns out that R can be replaced with a data structure based on balanced search trees, such that operation $eval(\Sigma, m, R, \mathcal{S})$ at Line 8 of Algorithm 4 costs $O(\log \sigma)$ amortized time. With this modification, Algorithm 4 runs in $O(n + \bar{r} \log \sigma)$ time.

Lemma 45 *Algorithm 4 can be implemented with $O(n + \bar{r} \log \sigma)$ running time and $O(n)$ words of space where n is the text length, $\bar{r}$ is the number of runs in $\text{BWT}(T^{rev})$, and σ is the alphabet size.*

Proof We will prove that $eval(\cdot)$ can be implemented with $O(\log \sigma)$ amortized running time. We maintain a balanced binary search tree (BST) in which each node has key $l = R[c].len$ for some $c \in \Sigma$. Note that the BST has at most σ nodes, thus the height is $O(\log \sigma)$. Each node stores such corresponding alphabet symbols c's, maintaining them using two doubly linked lists, one for active symbols (i.e., those with $R[c].active = true$), and the other for inactive symbols (i.e., those with $R[c].active = false$). For each symbol $c \in \Sigma$, we store $R[c].pos$ and $R[c].active$, as well as the pointer to the BST node in which c is stored so that $R[c].len$ can be retrieved by accessing c, and the pointer to the corresponding node of the linked list so that we can move symbols between different linked lists in $O(1)$ time.

We implement Line 11 of Algorithm 4 as follows. We move the linked-list node corresponding to symbol $\text{BWT}[i']$ into the linked list for active symbols of the BST node associated with l (after creating a new node if it does not exist) and update $R[c].pos$ and $R[c].active$ for $c = \text{BWT}[i']$ accordingly. This takes $O(\log \sigma)$ time.

Our optimization of Algorithm 5 then works as follows. We retrieve all nodes associated with lengths greater than l, which can be performed in $O(\log \sigma + k_{ret})$ time where k_{ret} is the number of retrieved nodes. We iterate all active symbols $c \in \Sigma$ stored in the retrieved nodes to add $R[c].pos$ to $\mathcal{S}$ in $O(k_{act})$ time where k_{act} is the

total number of active symbols to report. Then we concatenate all the linked lists in the retrieved nodes in $O(k_{ret})$ time, and append it to the linked list for inactive symbols in the node with key l (we create a new node if it does not exist), which takes $O(\log \sigma)$ time. Then we delete all the retrieved BST nodes (since they no longer contain any symbol) in $O(k_{ret} \log \sigma)$ time. As a consequence, Algorithm 5 runs in $O(k_{act} + (1 + k_{ret}) \log \sigma)$ time. Observe that at each c-run break, at most two symbols can become active and only $O(1)$ new BST nodes can be created. Therefore, the sum of $k_{ret} + k_{act}$ over all executions of $eval(\cdot)$ is $O(\bar{r})$. Therefore, the amortized running time is $O(\log \sigma)$. □

One-Pass Algorithm in Compressed Space

One important feature of our algorithm is that we only need one scan of the BWT, SA, and LCP to compute $\mathcal{S}$. This means that our algorithm also works in the streaming scenario where these arrays are provided one element at a time, from first to last. This feature can be exploited to run the algorithm in compressed working space using Prefix-free Parsing [33]. This optimization does not provide strong theoretical bounds, but in practice (see Section 8) it drastically reduces the amount of time and working space needed to compute the Suffixient Array.

Prefix-free parsing (PFP) is a technique introduced by Boucher et al. [33] to ease the computational burden of computing the BWT of large and repetitive texts. Briefly, with one linear-time scan of the text T, PFP divides T into overlapping segments, called *phrases*, of variable length, which are then used to construct what is referred to as the *dictionary* D (i.e., the set of distinct phrases) and *parse* P of the text (i.e., the text T encoded as a sequence of phrases, represented as indexes in D). Then, with a separate linear-time algorithm, the BWT of T is directly computed from D and P; thus using space proportional to the combined size $|D| + |P|$ of the two data structures. Very repetitive texts will tend to generate very small D, since text repetitions translate to repeated phrases. The size of P (number of phrases in which T is parsed), is instead controlled by a user-defined parameter and is a small fraction of T's length n. In [23, 46], it was shown how to modify PFP in order to compute also the SA and the LCP arrays. This version streams the three arrays BWT, LCP, and SA of T in $O(|D| + |P|)$ compressed space, from their first to last entry and in parallel (i.e., the triples (BWT[i], LCP[i], SA[i]) are output for $i = 1, 2, \ldots, n$); this is sufficient for running Algorithm 4 in compressed space, without affecting its running time. The only change consists in reading the text T backwards, in order to compute BWT(T^{rev}), LCP(T^{rev}), and SA(T^{rev}). If the input T resides on disk, this can be achieved very easily with a backward scan of T (without increasing the number of I/Os - page swaps - with respect to reading T forward). If, instead, the text is read from a forward stream, then PFP yields BWT(T), LCP(T), and SA(T) and our algorithm computes a suffixient set for T^{rev}. Our index still works, except that we have to reverse the query pattern before searching for it in the index. It is worth mentioning that it is actually also possible to modify PFP so that it streams BWT(T^{rev}), LCP(T^{rev}), and SA(T^{rev}) while reading a forward stream for T. However, in view of the simpler solutions described above, we do not enter into such details.

In Section 8 we will show that our PFP-based optimization makes it possible to compute the Suffixient Array efficiently on massive repetitive datasets.

6.3 A Linear-Time Algorithm using LF-Mapping

In this section, we further speed up the one-pass algorithm provided in the previous section and achieve linear time. Our new algorithm is summarized in Algorithm 6.

Algorithm 6 First linear-time algorithm building a smallest suffixient set.

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

$\mathcal{S} \leftarrow \emptyset$; /* initialize the empty suffixient set $\mathcal{S}$ */
BWT $\leftarrow$ BWT(T^{rev}); LCP $\leftarrow$ LCP(T^{rev}); SA $\leftarrow$ SA(T^{rev});
$R[1,\sigma] \leftarrow ((-1,0,\mathit{false}),\ldots,(-1,0,\mathit{false}))$; /* LCP local maxima */
LF$[1,\sigma] \leftarrow (0,\ldots,0)$; /* LF mapping */
$m \leftarrow \infty$; /* LCP minimum inside current BWT run */
for all $c = 2,\ldots,\sigma$: LF$[c] \leftarrow$ LF$[c-1] + occ(T, c-1)$;
LF[BWT[1]] $\leftarrow$ LF[BWT[1]] $+ 1$;
for $i = 2,\ldots,n$ **do**
  LF[BWT[i]] $\leftarrow$ LF[BWT[i]] $+ 1$;
  $m \leftarrow \min\{m, \text{LCP}[i]\}$;
  if BWT[i] $\neq$ BWT[$i-1$] **then**
    for $i' \in \{i-1, i\}$ **do**
      if $i' = i-1$ **then** $\mathit{eval}(\{\text{BWT}[i']\}, m, R, \mathcal{S})$;
      else if $R[\text{BWT}[i']].\mathit{len} \neq -1$ **then**
        $\mathit{eval}(\{\text{BWT}[i']\}, \text{LCP}[\text{LF}[\text{BWT}[i']]] - 1, R, \mathcal{S})$;
      end
      if LCP[i] $> R[\text{BWT}[i']].\mathit{len}$ **then**
        $R[\text{BWT}[i']] \leftarrow (\text{LCP}[i], n - \text{SA}[i'] + 1, \mathit{true})$;
      end
    end
    $m \leftarrow \infty$;
  end
end
$\mathit{eval}(\Sigma, -1, R, \mathcal{S})$; /* evaluate last active candidates */
return $\mathcal{S}$;

As previously discussed, in Algorithm 4, for the one-character extension of every BWT equal-letter run BWT$[i^*, i] = cc\ldots ccx$ (with $x \neq c$) we run procedure *eval* (Algorithm 5) to check if min LCP$[i^*+1, i]$ drops below the current LCP maxima associated to y-run breaks, for all $y \in \Sigma$. In the end, this step charges an additional $O(\bar{r}\sigma)$ term (which we mentioned can be reduced to $O(\bar{r}\log\sigma)$ by using opportune data structures). Intuitively, Algorithm 6 avoids this cost by calling the *eval* procedure only on the two candidates $R[c']$, where $c' \in \text{BWT}[i-1, i]$. We achieve this by introducing a new array LF$[1,\sigma]$ updated on-the-fly, implementing the *LF-mapping* property of the Burrows-Wheeler transform. More formally: assume we have scanned the BWT up to the c-run break i, and let $i' \in \{i-1, i\}$ be such that BWT$[i'] = c$. Then, LF is such that LF[BWT$[i']$] = LF$[c] = j$, with SA$[j]$ = SA$[i'] + 1$. In other words, BWT$[j]$ is the character preceding BWT$[i']$ in T^{rev}. Our linear-time algorithm

is based on the following idea. Assume for simplicity that i is not the first c-run break. Let $i^* < i$ be the largest integer such that i^* is a c-run break as well (i.e., i^* is the c-run break immediately preceding i). As in Algorithm 4, our new Algorithm 6 maintains $R[c].len = \text{LCP}[i^*]$. By Lemma 41, we have to discover if $i \in B_{i^*,c}$; if this is the case, then we compare $\text{LCP}[i]$ and $\text{LCP}[i^*]$ and decide if $\text{LCP}[i]$ is the new maximum in $B_{i^*,c} \cap [i]$ (i.e., $\text{LCP}[i] > \text{LCP}[i^*]$) or if the local maximum among c-run breaks in $B_{i^*,c} \cap [i]$ had already been found (i.e., $\text{LCP}[i] \leq \text{LCP}[i^*]$). If, on the other hand, $i \notin B_{i^*,c}$, then we insert in $\mathcal{S}$ the local maximum $R[c].pos$ relative to $B_{i^*,c}$ if and only if $R[c].active = true$ (if $R[c].active = false$, then the suffixient position corresponding to the local maximum of $B_{i^*,c}$ had already been stored in $\mathcal{S}$).

In order to discover if $i \in B_{i^*,c}$, we distinguish two cases.

(i) If $\text{BWT}[i-1] = c$ then, since i^* is the previous c-run break, it must be the case that $\text{BWT}[i^*-1, i^*, \ldots, i-1, i] = ycc\ldots ccx$, for some $x \neq c$ and $y \neq c$. It follows that $i \in B_{i^*,c}$ if and only if $\min \text{LCP}[i^*, i] \geq R[c].len = \text{LCP}[i^*]$. Algorithm 6 computes $\min \text{LCP}[i^*, i]$ analogously to Algorithm 4, i.e., by updating a variable m storing the minimum LCP inside intervals corresponding to BWT equal-letter runs.

(ii) If, on the other hand, $\text{BWT}[i-1] = x \neq c$, then since i^* is the previous c-run break, it must be that $\text{BWT}[i^*-1, i^*, \ldots, i-1, i] = cy\ldots xc$, for some $x \neq c$ and $y \neq c$ (and there are no other occurrences of c between $\text{BWT}[i^*-1]$ and $\text{BWT}[i]$). As in the previous case, the goal is to compute $\min \text{LCP}[i^*, i]$. We achieve this by using the LF array. Due to the way the array LF is defined and constructed, we know that $\min \text{LCP}[i^*, i] = \text{LCP}[\text{LF}[c]] - 1$. As a result, we have that $i \in B_{i^*,c}$ if and only if $\text{LCP}[\text{LF}[c]] - 1 \geq R[c].len = \text{LCP}[i^*]$.

From the above intuition, we obtain:

Lemma 46 *Given a text $T[1, n]$, Algorithm 6 computes a smallest suffixient set $\mathcal{S}$ in $O(n)$ time and $O(n)$ words of space.*

Proof Algorithm 6 scans the BWT exactly once and, for each position of it, performs $O(1)$ accesses to the SA and the LCP and calls Algorithm 5 at most once for each run break. Since Algorithm 5 performs a constant number of operations and the LF-mapping array can be computed in $O(n)$ time, the whole algorithm runs in $O(n)$ time. In addition, the only supplementary data structure we need is R, which consumes $O(\sigma)$ words of space; thus, altogether, we take $O(n + \sigma)$ words of space. Since we assumed $\sigma \leq n$, the space consumption of Algorithm 6 is $O(n)$ words.

We prove the correctness of Algorithm 6 by showing that it computes the same output as Algorithm 3. In particular, given $\mathcal{S}'$, the output of Algorithm 6, and $\mathcal{S}$, the output of Algorithm 3,

we show that for any value $s = n - \text{SA}[i] + 1$, where $i \in [n]$, *(1)* $s \in \mathcal{S} \implies s \in \mathcal{S}'$ and *(2)* $s \notin \mathcal{S} \implies s \notin \mathcal{S}'$.

(1) Let $(n - \text{SA}[i'] + 1) \in \mathcal{S}$, where i is a c-run break, such that $i' \in \{i-1, i\}$ and $\text{BWT}[i'] = c$. By Lemma 41, we have $\max \text{LCP}[B_{i,c}] = \text{LCP}[i]$. Assume we have scanned the BWT up to position i. There are three cases: i' is the position of the first occurrence of c in the BWT, or there exists another c-run break at position

$j < i$, such that either $i \in B_{j,c}$ or $box(i) \cap box(j) = \emptyset$. In all three cases, since $(n - \text{SA}[i'] + 1) \in \mathcal{S}$, we have $\text{LCP}[i] > R[c].len$; thus, i is set as the new active c candidate in R (lines 17-19). If i' is the position of the last occurrence of $\text{BWT}[i']$, since $R[c].active = true$, after traversing the whole BWT, Algorithm 6 inserts $R[c].pos = (n - \text{SA}[i] + 1)$ in $\mathcal{S}'$ (line 4, Algorithm 5). Otherwise, let $i < k'$ be the position of the next occurrence of c in the BWT, such that $k' \in \{k-1, k\}$. Due to $\max \text{LCP}[B_{i,c}] = \text{LCP}[i]$, it must hold $\text{LCP}[k] < \text{LCP}[i]$ or $k \notin box(i)$. Here either, $\text{BWT}[i-1, k'] = xcc...ccy$ where $x \neq c$ and $y \neq c$; thus, $\exists l \in \text{LCP}[i+1, k']$ such that $l < R[c].len$, or $\text{LCP}[\text{LF}[c]]-1 < R[c].len$. Again, since $R[c].active = true$, then Algorithm 6 insert $R[c].pos = (n - \text{SA}[i] + 1)$ in $\mathcal{S}'$ (line 4, Algorithm 5).

(2) Now let $(n-\text{SA}[i']+1) \notin \mathcal{S}$, this means that $\exists j \in B_{i,c}$ such that $\text{LCP}[j] > \text{LCP}[i]$ and j is a c-run break. We need to consider two cases: *(i)* $j > i$ and *(ii)* $j < i$. *(i)* If $j > i, \forall l \in \text{LCP}[i+1, j], l \geq \text{LCP}[i]$, which implies $\text{LCP}[\text{LF}[c]] - 1 \geq \text{LCP}[i]$ for all c-run breaks in $[i+1, j]$; thus, we never update $R[c]$ to the inactive state (lines 2-6, Algorithm 5) until we scan position j. Here, we get $\text{LCP}[j] > R[c].len$ and update $R[c].pos = (n - \text{SA}[j'] + 1)$ (lines 17-19), where $j' \in \{j-1, j\}$ and $\text{BWT}[j'] = c$. Due to this, $(n - \text{SA}[i'] + 1)$ is dropped and not inserted in $\mathcal{S}'$. *(ii)* If $j < i$, then $\exists l \in [j+1, i]$ such that $\text{LCP}[\text{LF}[c]] - 1 < R[c].len$ or $\text{LCP}[l] < \text{LCP}[i]$; thus, when we read l, $R[c]$ is updated to the inactive state (line 6, Algorithm 5). However, since $\forall s \in B_{i,c} \cap [j+1, i], \text{LCP}[s] < \text{LCP}[j]$ it means that $R[c]$ is never updated to the active state for any position in $[j+1, i]$ (we skip lines 17-19), so also in this case $(n - \text{SA}[i'] + 1)$ is not inserted in $\mathcal{S}'$.

□

In Fig. 4 we show an example of how Algorithm 6 works.

6.4 Another Linear-Time Algorithm via Precomputing Boxes

In this section we propose a second linear algorithm which in practice is faster than Algorithm 6, at the cost of increasing by an additive term $O(n)$ the space consumption. In Section 8 we show our experimental results comparing both algorithms.

For a given c-run break i_f for $f \in [k]$, let $i'_f \in \{i_f - 1, i_f\}$ be such that $\text{BWT}[i'_f] = c$. Algorithms 4 and 6 will add to the suffixient set the position $n - \text{SA}[i'_f] + 1$ if and only if

$$i_f = \min\{i \ : \ i \in B_{i_f,c} \wedge \text{LCP}[i] = \max \text{LCP}[B_{i_f,c}]\}. \tag{1}$$

In the following, we expose a method to evaluate (1) in $O(1)$ time for each $i \in [n]$, which will lead to a linear-time algorithm. As a first step, we define *first c-maximum* positions (Definition 47). Later, we prove that those positions are exactly the positions satisfying (1) (Proposition 48). Finally, we show an algorithm computing a suffixient set of smallest cardinality by finding *first c-maximum* positions (Algorithm 7).

Definition 47 (*first c-maximum*) Let i_f the position of a c-run break. We say i_f is *first c-candidate* if one of the following conditions hold:

- $f = 1$, or

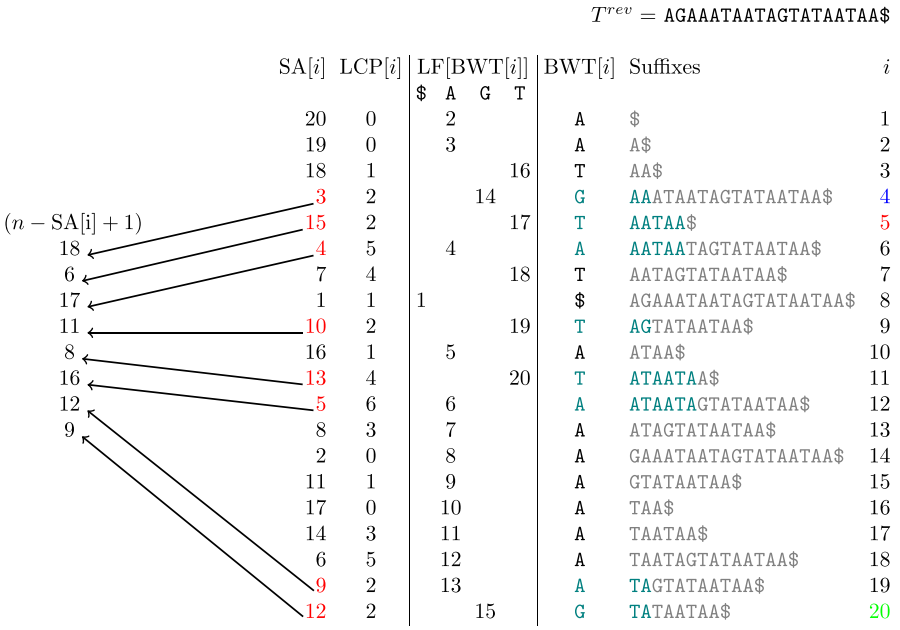

Fig. 4 The figure shows the data used by Algorithm 6 to construct a smallest suffixient set $\mathcal{S}$ for a text $T[1, n]$ with $n = 20$. Extending Fig. 3, we include the LF array with four additional columns representing how LF[c] is updated over i for each $c \in \Sigma$. The i-th row in these columns indicates the result obtained when updating LF[BWT[i]] = LF[BWT[i]] + 1 in line 9 of Algorithm 6. For brevity, we show how Algorithm 6 works only on the G-run breaks. Consider the first G-run break at position $i = 4$ (highlighted in blue in column i). At this stage we have LF[G] = $occ(T, \$) + occ(T, \texttt{A}) = 13$ and $R[\texttt{G}] = (-1, 0, \mathit{false})$, so we update LF[G] = LF[G]+1 = 14 (line 9). Since BWT[i] = G (then $i' = i$) and $R[\texttt{G}].len = -1$, we do not call `eval` on either line 13 or line 14, so $R[\texttt{G}].pos$ is not added to $\mathcal{S}$. Now, we have LCP[4] = 2 > $R[\texttt{G}].len = -1$, then we set $R[\texttt{G}] = (\text{LCP}[4], n - \text{SA}[4] + 1, \mathit{true}) = (2, 18, \mathit{true})$ (line 18) and $m = \infty$ (line 21). For the next G-run break at position $i = 5$ (highlighted in red in column i), since we set $m = \min(\text{LCP}[5], \infty) = 2$ and LCP[LF[G]] − 1 = −1 < $R[\texttt{G}].len = 2$, we do not add $R[G].pos$ to $\mathcal{S}$ in either line 13 nor line 14. Next, since LCP[5] = 2 = $R[\texttt{G}].len$, we do not update $R[\texttt{G}]$ on line 18, and we finish this iteration. The next G-run break occurs at position $i = 20$ (highlighted in green in column i). We set LF[G] = LF[G] + 1 = 15 and $m = \text{LCP}[20] = 2$. Since $i' = i$, we do not call `eval` on line 13, but since LCP[LF[G]] − 1 = 0 < $R[\texttt{G}].len = 2$, we add $R[\texttt{G}].pos = 18$ to $\mathcal{S}$ and we set $R[\texttt{G}] = (0, 0, \mathit{false})$ on the `eval` calling on line 15. Next, we have $R[\texttt{G}].len = 0 < \text{LCP}[20] = 2$, so we update $R[\texttt{G}] = (2, 9, \mathit{true})$. Finally, since $R[\texttt{G}].active = \mathit{true}$, we add $R[\texttt{G}].pos = 9$ on the final `eval` calling on line 24

- $i_{f-1} \leq l_f - 1$

Moreover, we say that i_f is *first c-maximum* if it is *first c-candidate* and one of the following conditions hold:

- $f = k$, or
- for each $f < j \leq k$, if $i_f \leq l_j - 1$ then $r_f + 1 < i_j$.

Intuitively, *first c-candidate* positions are those such that its LCP value is smaller than the LCP of the previous c-run break. On the other hand, *first c-maximum* positions are *first c-candidate* positions whose box only intersects c-run break positions not having larger LCP values. The following proposition establishes the equivalency

between finding c-run breaks which are the leftmost local maxima within a box and finding *first c-maximum* positions.

Proposition 48 *Let i_f be a c-run break. We have*

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

if and only if i_f is first c-maximum.

Proof Let $B_{i_f,c} = \{i_{f-a}, \ldots, i_f, \ldots, i_{f+b}\}$. By definition it holds for every $j \in [f-a, f+b]$ that $\mathrm{LCP}[i_j] \geq \mathrm{LCP}[i_f]$, and it also holds that

$$i_{f-a-1} < l_f \leq i_{f-a} < \cdots < i_f < \cdots < i_{f+b} \leq r_f < i_{f+b+1} \tag{2}$$

where $[l_f, r_f] = box(i_f)$ and we define $i_0 := 0$ and $i_{k+1} := n+1$ for brevity.
($\Rightarrow$) Suppose

$$i_f = \min\{i \ : \ i \in B_{i_f,c} \wedge \mathrm{LCP}[i] = \max \mathrm{LCP}[B_{i_f,c}]\}.$$

Then by the maximality of $\mathrm{LCP}[i_f]$, it holds that $\mathrm{LCP}[i_f] = \mathrm{LCP}[i']$ for all $i' \in B_{i_f,c}$. This implies that $a = 0$ (due to the minimality of i_f) and $box(i_f) = \cdots = box(i_{f+b}) = [l_f, r_f]$. Since it holds $i_{f-1} \leq l_f - 1 < l_f \leq i_f$ by (2), i_f is *first c-candidate*. To prove that i_f is also *first c-maximum*, we need to prove that it holds for every $f < j \leq k$ if $i_f \leq l_j - 1$ then $r_j + 1 < i_j$ where $[l_j, r_j] = box(i_j)$. For $f < j \leq f+b$, observe that $l_j - 1 = l_f - 1 < i_f$, and the implication is true because the premise is false. Now consider $f+b < j \leq k$. Then it holds that $r_f < r_f + 1 \leq i_j$ by (2). We have two cases: (i) $r_f + 1 < i_j$ and (ii) $r_f + 1 = i_j$. For the former case, we are done. For the latter case, by definition of $box(i_f)$, it holds that $\mathrm{LCP}[h] \geq \mathrm{LCP}[i_f] > \mathrm{LCP}[i_j]$ for every $h \in [l_f, r_f]$, which implies that $l_j \leq l_f \leq i_f$. Since $l_j - 1 < l_j \leq i_f$, the implication is true due to the false premise. Therefore, i_f is *first c-maximum*.
($\Leftarrow$) We prove by contrapositive. Suppose

$$i_f \neq i_m = \min\{i \ : \ i \in B_{i_f,c} \wedge \mathrm{LCP}[i] = \max \mathrm{LCP}[B_{i_f,c}]\}.$$

If $f < m$, then $b \geq 1$ and, since $i_m \in B_{i_f,c}$, $\mathrm{LCP}[i_f] < \mathrm{LCP}[i_m]$, we have that $i_f \leq l_m - 1 < l_m$. However, it holds that $i_f < i_m \leq i_{f+b} \leq r_f < r_f + 1$ by (2). Since it holds that (i) $f < m \leq k$ and (ii) $i_f < l_m - 1$ and (iii) $r_f + 1 > i_m$, it follows that i_f cannot be *first c-maximum*. On the other hand, if $m < f$, then $a \geq 1$. From $l_f \leq i_{f-a}$ in (2), it follows that $l_f - 1 < l_f \leq i_{f-a} \leq i_{f-1}$. Therefore, i_f cannot be *first c-candidate*. □

As shown in the proof of Proposition 48, if i_f is *first c-candidate* and there exists a position $i_f < i_e$ such that $l_{f+1} - 1 < i_f, \ldots, l_{e-1} - 1 < i_f$, $i_f \leq l_e - 1$ and $r_f + 1 < i_e$, then for any other position $i_e < i_g$ such that $i_f \leq l_g - 1$ (if any exist) it holds $r_f + 1 < i_g$. Indeed, i_e is also *first c-candidate*, which means that, to find *first c-maximum* positions, we just need to sequentially scan the BWT looking for *first*

Algorithm 7 Second linear-time algorithm building a smallest suffixient set.

```
   input : A text T[1, n] over a finite alphabet Σ.
   output: A smallest suffixient set for T.
 1 S ← ∅;
 2 BWT ← BWT(T^rev); LCP ← LCP(T^rev); SA ← SA(T^rev);
 3 NSV ← NSV(LCP); PSV ← PSV(LCP);
 4 R[1, σ] ← ((sa_pos ← 0, text_pos ← 0, active ← false, nsv ← n + 1) × σ);
 5 for i = 2, ..., n do
 6     if BWT[i] ≠ BWT[i − 1] then
 7         for i′ ∈ {i − 1, i} do
 8             if R[BWT[i′]].sa_pos ≤ PSV[i] then
 9                 if R[BWT[i′]].nsv < i then
10                     S ← S ∪ {R[BWT[i′]].text_pos};
11                 end
12                 R[BWT[i′]] ← (i, n − SA[i′] + 1, true, NSV[i]) ;
13             end
14         end
15     end
16 end
17 foreach c ∈ Σ do
18     if R[c].active = true then S ← S ∪ {R[c].text_pos};
19 end
20 return S;
```

c-candidate positions and compare their boxes' boundaries. Consider the two arrays defined based on the LCP array as the following.

Definition 49 (PSV/NSV arrays) For a given length-n integer array $A[1, n]$, the *previous smaller value* array of A is an integer array of length n defined as $\text{PSV}(A)[i] = \max(\{j \mid j < i, A[j] < A[i]\} \cup \{0\})$ for all $i \in [n]$. In a similar way, the *next smaller value* array of A is defined as $\text{NSV}(A)[i] = \min(\{j \mid j > i, A[j] < A[i]\} \cup \{n + 1\})$ for all $i \in [n]$.

Interestingly, if we have access to the PSV(LCP) and NSV(LCP) arrays, we can compute $box(i) = [l_i, r_i] = [\text{PSV(LCP)}[i] + 1, \text{NSV(LCP)}[i] - 1]$ in $O(1)$ time. These arrays can be computed in $O(n)$ time using a stack-based algorithm [48] and one can also use space-efficient data structures (e.g., [47]) if a smaller working space is of interest. For simplicity, in the rest of this section PSV, NSV denote PSV(LCP), and NSV(LCP), respectively.

Following the above ideas, we show in Algorithm 7 a procedure adding to the suffixient set *first c-maximum* positions. In the same spirit as Algorithms 4 and 6, $R[c]$ keeps the information of the last c-run break found up to now which is *first c-candidate*. The algorithm sequentially scans the BWT looking for run breaks. For each c-run break at position i, if i is *first c-candidate*, $R[c]$ is updated as $(sa\_pos \leftarrow i, text\_pos \leftarrow n - \text{SA}[i'] + 1, active \leftarrow true, nsv \leftarrow \text{NSV}[i])$. Whenever $R[c]$ is being updated, we check if the last *first c-candidate* is *first c-maximum*, and report it if so. We conclude this section with the following.

Lemma 50 *Given a text $T[1, n]$ over alphabet of size σ, Algorithm 7 computes a smallest suffixient set $\mathcal{S}$ in $O(n)$ time and $O(n)$ words of space.*

Proof We prove the correctness by showing the following invariant: After we scanned the BWT up to position i, it holds for every $c \in \Sigma$ that (i) $R[c]$ has the information of the last c-run break found up to position i which is *first c-candidate* unless c has not appeared yet, and (ii) all *first c-maximum* positions i' with $\mathrm{NSV}[i'] < i$ have been reported.

Since $R[c]$ is set to $(0, 0, \mathit{false}, n+1)$ at the beginning of the Algorithm (line 4) for each $c \in \Sigma$, the first c-run break always satisfies the condition in line 8, so $R[c]$ is updated and set as *first c-candidate* in line 12. This is correct because the first c-run break is always *first c-candidate*. Note that the condition in line 9 is not satisfied for the first c-run break, so it does not report anything. Therefore after processing the first c-run break, the invariant holds.

Now assume we have scanned the BWT until position $i-1$ and suppose i is a c-run break, i.e., $\mathrm{BWT}[i'] = c$ with $i' \in \{i-1, i\}$. Also, assume that $j(<i)$ is the previous c-run break which is *first c-candidate*, thus $\mathrm{BWT}[j'] = c$ with $j' \in \{j-1, j\}$. Then $R[c].sa\_pos = j$, $R[c].text\_pos = n - \mathrm{SA}[j'] + 1$, $R[c].active = \mathit{true}$, and $R[c].nsv = \mathrm{NSV}[j]$. If $R[c].sa\_pos \leq \mathrm{PSV}[i]$ (line 8), then we know i is *first c-candidate* and we update the attributes of $R[c]$ according of i's values: $R[c].sa\_pos \leftarrow i$, $R[c].text\_pos \leftarrow n - \mathrm{SA}[i'] + 1$, $R[c].active \leftarrow \mathit{true}$, and $R[c].nsv \leftarrow \mathrm{NSV}[i]$; hence Invariant (i) holds after processing position i. Before updating $R[c]$, we check if $R[c].nsv < i$. If it is satisfied, j is *first c-maximum* and we add $R[c].text\_pos$ to the suffixient set built up to now (line 9), with which Invariant (ii) holds. After processing all positions, by the invariants, for every $c \in \Sigma$ that occurs in T, $R[c]$ has the last c-run break which is *first c-candidate*, which is *first c-maximum* by definition), so we add it to the outputted suffixient set in line 17. Note that for every $c \in \Sigma$, $R[c].active$ is set to *true* if and only if c occurs in T because it is set to *true* at the first c-break and remains *true*, which ensures that we do not add $R[c].text\_pos$ to the suffixient set if c does not appear in T.

As far as the running time is concerned, computing BWT, LCP, and SA on line 2 takes $O(n)$ time. In addition, computing PSV and NSV on line 3 also takes $O(n)$ time using a stack-based algorithm as mentioned earlier. Scanning the BWT takes $O(n)$ time, and it is performed $O(1)$ operations for each position of the BWT. Then, the total running time is $O(n)$. □

In Fig. 5 we show an example of how Algorithm 7 works.

7 Implementation Details

We discuss two optimizations to Algorithm 1 that in practice are expected to lower the I/O complexity to near-optimal. These optimizations have been included in our implementation, which in the next section we compare with the r-index. Our optimizations are motivated by a particular application: pattern matching on repetitive genomic collections (in particular, in this section we will assume $\sigma \in O(1)$). As discussed in the introduction, a lot of interest has recently been dedicated to solving pattern matching queries of short DNA fragments (m is of the order of thousands of characters) on collections composed of several genomes from the same species. Since genomes within

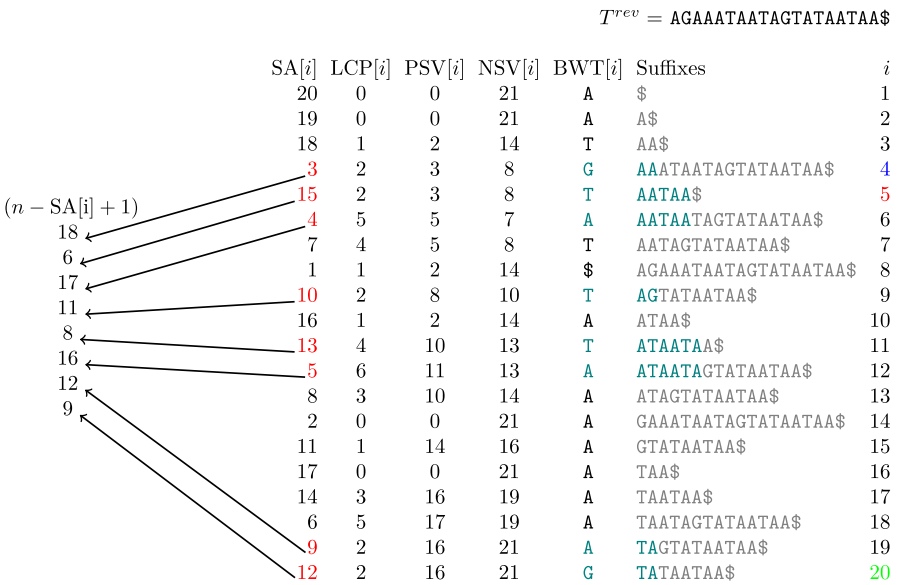

Fig. 5 The figure shows the data used by Algorithm 7 to construct a smallest suffixient set $\mathcal{S}$ for a text $T[1, n]$ with $n = 20$. In addition to the data shown in Fig. 3, here we include the PSV and NSV arrays. For brevity, we show how Algorithm 7 works only on the G-run breaks. Consider the first G-run break at position $i = 4$ (highlighted in blue on column i). At this stage we have $i' = i$ and $R[\mathtt{G}] = (0, 0, \mathit{false}, 21)$. Since $R[\mathtt{G}].sa\_pos = 0 < 3 = \text{PSV}[4]$ and $i = 4 < 21 = R[\mathtt{G}].nsv$, then condition on line 8 is satisfied but condition on line 9 is not, so we do not add $R[\mathtt{G}].text\_pos$ to $\mathcal{S}$ in line 10 and we update $R[\mathtt{G}] = (i, n - \text{SA}[i'] + 1, \mathit{true}, \text{NSV}[i]) = (4, 18, \mathit{true}, 8)$ on line 12. For the next G-run break at position $i = 5$ (highlighted in red on column i), we have $\text{PSV}[4] = 3 < 4 = R[\mathtt{G}].sa\_pos$, then condition on line 8 is not satisfied and nothing else is done in this iteration. The next G-run break occurs at position $i = 20$ (highlighted in red on column i). We have $i' = i$. Since $R[\mathtt{G}].sa\_pos = 4 < 16 = \text{PSV}[20]$ and $R[\mathtt{G}].nsv = 8 < 20 = i$, we add $R[\mathtt{G}].text\_pos = 18$ to $\mathcal{S}$ on line 10 and we update $R[\mathtt{G}] = (i, n - \text{SA}[i'] + 1, \mathit{true}, \text{NSV}[i]) = (20, 9, \mathit{true}, 21)$ on line 12. Finally, since $R[\mathtt{G}].active = \mathit{true}$, we add $R[\mathtt{G}].pos = 9$ on the final `foreach` loop on line 18

those collections have $\geq 99.9\%$ similarity, repetitive genomic collections tend to be extremely repetitive (and compressible).

Optimization I

We start from the version of Algorithm 1 that implements $search()$ with binary search on sA and with random access on the oracle as described in Theorem 19. The first optimization that we introduce is that we start the `while` loop of Algorithm 1 from $i = \min\{m, k\}$, where $k = \lceil \log_\sigma \chi \rceil + O(1)$ is a parameter whose value is justified below (paragraph *Optimization II*). Intuitively, this optimization is motivated by the fact that we expect all short (length $< k$) prefixes of P to be right-maximal with large probability. This implies that those prefixes will trigger many useless binary searches in Algorithm 1[4]. If the first binary search with the above value of i finds a match of length i, our search procedure proceeds normally as described in Algorithm 1.

[4] As a matter of fact, only one binary search is really useful: the one for the longest pattern prefix $P[1, i]$ suffixing $T[1, x]$ for some $x \in$ sA. Since, however, we do not know i in advance, our algorithm needs to run several binary searches on different pattern prefixes before finding i.

Otherwise, we decrement i by one unit and repeat in order to find the longest prefix of $P[1, k]$ suffixing $T[1, x]$ for some $x \in$ sA. It is worth mentioning that in practice the number of binary searches dramatically decreases only with this optimization; for example, for patterns of length 10,000 in the DNA dataset we used in the experiments, it was observed that less than four binary searches are sufficient on average.

Optimization II

Let $k = \lceil \log_\sigma \chi \rceil + O(1)$ be the same parameter used in the above optimization. For every $x \in$ sA, we bit-pack $T[x - k + 1, x]^{\text{rev}}$ in an integer of $k\lceil \log_2 \sigma \rceil$ bits. Since we assume $\sigma \in O(1)$, each such integer is bounded by $O(\sigma^k) = O(\chi)$. Let L be the list, sorted in ascending order, containing such integers. L is encoded succinctly using an Elias-Fano predecessor data structure (see for example [49]; in our code we employ the `sd_vector` implementation of `sdsl-lite`[5]), using $\chi \log(\sigma^k/\chi) + O(\chi) = O(\chi)$ bits, which add only low-order terms on top of the space ($\chi \log n$ bits) of the Suffixient Array.

The implementation of $search(\beta)$ is modified as follows. If $LCS(\beta, T[1, x]) < k$ for all $x \in$ sA, then L suffices to answer $search(\beta)$ with just one predecessor query on the integer formed by packing the characters of β^{rev}. Otherwise ($LCS(\beta, T[1, x]) \geq k$ for some $x \in$ sA), a predecessor query on the last k characters of β yields the range on sA corresponding to prefixes $T[1, x]$ ($x \in$ sA) suffixed by $\beta[|\beta| - k + 1, |\beta|]$. This range is then refined with the remaining characters of β via standard binary search and random access on T. For the chosen value of k, the Elias-Fano data structure answers predecessor queries in $O(\log(\sigma^k/\chi)) = O(1)$ time.

Recall that, by Theorem 19, our index has worst-case $O((1 + m/B) \cdot d \log \chi)$ I/O complexity, where $d \leq m$ is the node depth of the locus of P in the suffix tree of T (that is, the number of suffix tree edges traversed while searching P in the suffix tree of T). In our application (repetitive genomic collections), Optimization I is expected to drastically reduce the effect of the term d. Intuitively, this happens because Optimization I allows us to skip k levels of the Suffix Tree of T in constant time, therefore d is replaced by the number d' of edges that we traverse in the suffix tree starting from the locus of $P[1, k]$. A closer analysis shows that, rather than considering the full suffix tree, the same observation holds on the sparse suffix tree containing only the χ text suffixes starting in $j + 1$ for $j \in$ sA. To see this, observe that Algorithm 1 always skips the longest common prefix of $T[j + 1..]$ and $P[i + 1..]$ (by incrementing i and j in Line 3) for j that always belongs to sA. But then, we expect that the subtree (of the above sparse suffix tree) rooted in the locus of $P[1, k]$ contains very few leaves (if the individual genomes were really uniform, we would expect this number of leaves to be constant with large probability). In turn, d' is upper-bounded by such number of leaves.

Optimization II, on the other hand, is expected to reduce the term $\log \chi$. The intuitive reason is that list L defined above contains χ integers. Since (by Lemma 11) sA forms a small string attractor and individual genomes in repetitive collections have high entropy, we expect L to have high entropy as well. Therefore, we expect that the seeding strategy implemented in Optimization II drastically reduces the binary search

[5] https://github.com/simongog/sdsl-lite/blob/master/include/sdsl/sd_vector.hpp

range from χ to a very small length (in fact, this value would be constant in expectation if genomes were really uniform strings).

8 Experimental Results

We implemented all construction algorithms of Section 6 and the suffixient-based index of Theorem 18 (in two variants, read below) in C++ and made the code publicly available at https://github.com/regindex/suffixient-array.

We divide our experimental evaluation into four parts: first, we empirically study the new repetitiveness measure χ by comparing it with $\bar{r}$ under different alphabet orderings. Then, we assess the performance of our PFP-based one-pass algorithm, Algorithm 4 from Section 6.2, on massive genomics datasets. Then, we compare the other construction algorithms (requiring much more working space) on smaller datasets. Finally, we compare the performance of our suffixient-based index with those of the Prefix Array and the r-index [18] on the task of locating one pattern occurrence, using three random access oracles (for both our index and the Prefix Array).

We ran our experiments using two workstations: the first is an Intel(R) Xeon(R) W-2245 CPU @ 3.90GHz with 8 cores and 128 gigabytes of RAM running Ubuntu 18.04 LTS 64-bit. This workstation was used for the experiments on repetitiveness measures (Section 8.1), on sA construction on massive datasets (Section 8.2), and on indexing (Section 8.4). The second workstation is an 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. This workstation was used for the experiments on building Suffixient Arrays (Section 8.3), which required more working memory. We recorded the runtime and memory usage of our software by using the wall clock time and maximum resident set size from `/usr/bin/time` and two C++ libraries: `chrono` and `malloc_count`. The experiment was conducted and measured only once unless specified; however the results were sufficiently clear to support the conclusion.

8.1 The Repetitiveness Measure χ

In this experiment, we use nine biological datasets divided into two corpora. The first corpus contains six datasets containing DNA sequences from the Pizza&Chilli collection[6], while the second contains three datasets made by concatenating genomic sequences of different types. The three datasets of the second corpus are formed by 36,000 *SARS-CoV-2* assembled viral genomes, 220 *Salmonella enterica* assembled bacterial genomes, and 19 copies of the human *Chromosome 19*, respectively. We downloaded data from different sources: the viral genomes from the COVID-19 Data Portal[7], the bacterial genomes from NCBI[8], while the Chromosome 19 sequences belong to the 1,000 Genome project[9]. All datasets are filtered to keep only DNA

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

[7] https://www.covid19dataportal.org

[8] https://www.ncbi.nlm.nih.gov/assembly/?term=Salmonella+enterica

[9] https://github.com/koeppl/phoni

nucleotide characters A, C, G, T, in addition to the string terminator \$. As a consequence, the alphabet size in our experiments was $\sigma = 5$.

We compare the behavior of our new measure χ with the well-known measure $\bar{r}$: the number of equal-letter runs in the BWT of the reversed text. Since, unlike χ, measure $\bar{r}$ depends on a specific alphabet ordering, we explicitly compute $\bar{r}$ for all $(\sigma - 1)! = 24$ possible orderings (character \$ is always required to be the lexicographically-smallest one). Table 1 shows our results. Among all possible alphabet orderings, we only show the default one A < C < G < T, and the ones yielding the largest and smallest $\bar{r}$.

We observe that, in practice, χ is very close to (and always smaller than) $\bar{r}$ under any alphabet ordering; notice that theory only predicts $\chi \leq 2\bar{r}$ (Lemma 10). In this experiment, the minimum of $\bar{r}$ is always between 1.13 (dna) and 1.33 (Influenza) times larger than χ. We also observe that, while the alphabet ordering does not have a big influence on $\bar{r}$, the default alphabet ordering never yields the smallest $\bar{r}$. This suggests experimentally that χ is a better repetitiveness measure than $\bar{r}$, since it is not affected by the alphabet order (while always being close to $\bar{r}$).

8.2 Building Suffixient Arrays of Massive Datasets

In this experiment, we tested our PFP-based one-pass algorithm on two large genomic datasets: the first dataset is formed by 1,000 human Chromosome 19 copies (59GB), while the second contains 2,005,773 Sars-CoV-2 viral sequences (60GB).

As discussed in Section 6.2, we read the text T backwards from the disk keeping in RAM only the PFP data structures. We employed the PFP implementation provided in the `pscan.cpp` file contained in `Big-BWT` (https://github.com/alshai/Big-BWT) and ran it using 16 threads.

In both datasets, our algorithm performed very well with a maximum resident set size of 20.6 GB and 29.5 GB and a wall clock time of 52:19 (min:sec) and 2:06:37 (h:min:sec), on Chromosome 19 and Sars-CoV-2, respectively. The throughputs are 18.8MB/s and 7.9MB/s, respectively. It is also worth mentioning that the construction

Table 1 Summary of the results on the nine datasets. From left to right, we report the corpus name, the dataset name, the dataset length (number of characters), the size of the smallest suffixient set (χ), and the number of runs of the BWT of the reversed text ($\bar{r}$) for three different alphabet ordering: the default lexicographic order, and the two orderings leading to the minimum and maximum $\bar{r}$

corpus	dataset	dataset length	χ	default $\bar{r}$	min. $\bar{r}$	max. $\bar{r}$
Pizza&Chili	Cere	428,111,842	9,945,553	11,556,075	11,512,279	11,578,575
	E. Coli	112,682,447	13,126,726	15,041,926	14,974,687	15,043,686
	Influenza	154,795,431	2,234,085	3,015,173	2,979,820	3,023,774
	Para	412,279,605	13,399,378	15,581,823	15,475,176	15,635,533
	dna	403,920,884	215,201,641	243,480,086	243,179,189	243,574,132
	dna.001.1	104,857,499	1,414,711	1,717,155	1,715,622	1,718,418
	Chrom. 19	1,060,302,648	28,265,608	32,501,979	32,413,939	32,516,882
biological	Salmonella	1,043,921,520	19,070,836	22,407,573	22,254,453	22,416,480
	SarsCov2	1,052,893,440	829,777	1,091,476	1,088,925	1,093,104

time was mainly dominated by the time required to run the PFP preprocessing (11-19% of the total) and stream the SA, LCP, BWT arrays (76-86% of the total).

8.3 Other Suffixient Array Construction Algorithms

In this experiment, we measured the wall clock time and internal memory peak, as reported by the `usr/bin/time` utility, needed by the four Suffixient Array construction algorithms presented in Section 6: the one-pass algorithm (`one-pass`) using the non-compressed SA, LCP, BWT arrays, the PFP-based one-pass algorithm (`pfp`), and the first (`linear`) and second (`fm`) linear time algorithms. As input for the algorithms, we used several prefixes of the Sars-Cov-2 dataset described in Section 8.2. The i-th prefix of such dataset was created by concatenating the first $m \in \{2^i : i \in [14, 19]\}$ genomic sequences from the original dataset.

In Fig. 6 we provide a summary of the experimental results. We note that `pfp` is always faster and uses less space than the other three algorithms. In particular, on average, this algorithm was 3.74 times faster than `fm`, 5.59 times faster than `linear`, and 4.66 times faster than `one-pass`. At the same time, `pfp` used 57.96 times less space than `fm` and 20.56 times less space than `linear` and `one-pass`. This is due to the PFP preprocessing which exploits natural repetitiveness of genomic data to reduce time/space execution requirements. The `fm` algorithm was always faster than the other non-PFP-based algorithms. More in detail, `fm` was on average 1.41 times faster than `linear` and it was 1.18 times faster than `one-pass`. On the other hand, `fm` used much more working space than `linear` (3.03 times), to the point that it could not finish the computation for the largest prefix containing 2^{19} sequences due to memory requirements exceeding the available 256 GB of RAM. Interestingly, the (non-PFP based) `one-pass` algorithm was, on average, 1.2 times faster than `linear` while using the same amount of space. This could be explained given that, for constant alphabets (as the one of the Sars-Cov-2 dataset) the time complexity of the one-pass algorithm becomes $O(n)$. In addition, `one-pass` runs in a more cache-friendly way than `linear` since it only performs one linear scan of BWT, SA and LCP and

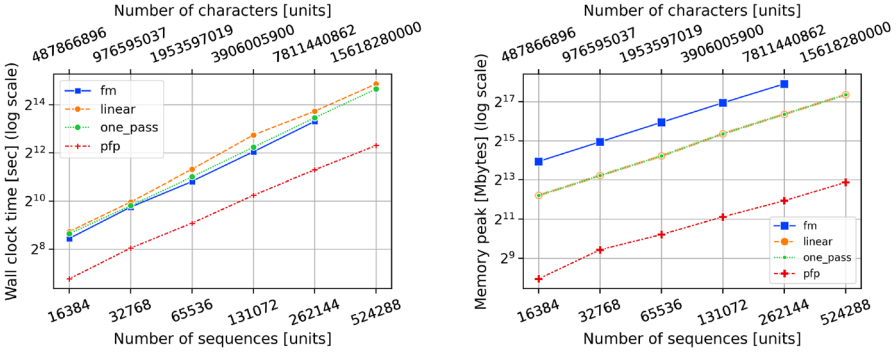

Fig. 6 Wall clock time (left) and memory peak (right) for computing a smallest suffixient set for six SarsCoV2 datasets using the four algorithms presented in Section 6. We generate the datasets by taking prefixes of increasing size containing $2^i : i \in [14, 19]$ genomic sequences

$O(\bar{r})$ linear scans of R containing σ one-character right-maximal extension candidates.

As a preliminary experiment to demonstrate this cache-efficiency, we separately tested the two algorithms on randomly generated repetitive strings of length $64 \cdot 2^{20}$ over alphabets with various sizes $2^2, \cdots, 2^{16}$. We observed that `one-pass` was faster than `linear` unless the alphabet size is greater than 2^{12}. We also measured the number of cache misses using the `perf` tool, and `one-pass` caused about 40% less cache misses than `linear`.

8.4 Pattern Matching Queries

In this last experiment, we compare our compressed index based on the Suffixient Array described in Section 4.1 with the Prefix Array and with the r-index of Gagie et al. [18] on the task of locating a single pattern occurrence.

8.4.1 Indexes and Random Access Oracles

We tested four indexes: two Suffixient-Array based indexes (sA and opt-sA detailed below), the Prefix Array (PA), and the toehold lemma of the r-index (`toehold`), i.e., the subset of the r-index sufficient to locate one pattern occurrence. In the plots, we also show the size of the full r-index for a comparison.

Random Access Oracles

The Suffixient-Array based indexes (sA and opt-sA) and the Prefix Array (PA) have been tested in combination with three random access oracles: the `bitpacked` text (a plain text representation using 2 bits per character), the subset `lz77` of the Lempel-Ziv 77 index of Kreft and Navarro[10] sufficient to perform random access (i.e., from the original index we removed all data structures needed for locating patterns), and an ad-hoc optimized implementation of Relative Lempel-Ziv [29] (`rlz`). The latter random access data structure: (i) packs the reference characters using 2 bits per character, (ii) automatically detects the best prefix of the input string to be used as the reference optimizing the overall compression ratio (trying lengths being a power of $(1+\epsilon)$, for a small $\epsilon > 0$), and (iii) optimizes the extraction of long substrings by accessing the predecessor structure (storing phrase borders) only when needed, i.e., a number of times equal to the number of phrases overlapping with the string to be extracted.

Suffixient Array-based Indexes (sA and `opt`-sA)

We implemented two variants of our suffixient-based index and combined them with the three random access oracles described above. The two variants implement the Optimizations described in Section 7. The first variant, labeled sA in the plots below, implements only Optimization I. The second variant, deemed opt-sA, implements both Optimizations I and II. Note that we implemented these optimizations assuming DNA alphabet, i.e., $\sigma = 4$, and we always set $k = 14$.

[10] https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/LZ

Prefix Array index (PA)
This index implements classic binary search on the Prefix Array, combining it with the three random access oracles described above.

Toehold Lemma of the r-Index (`toehold`)
We used the subset of the r-index [18] (we employ the original C++ implementation[11]) sufficient to locate one pattern occurrence (the so-called *Toehold lemma*); from the original implementation, we removed all data structures required to locate all other pattern occurrences. This made the index substantially smaller (compared to the full r-index). To ease reproducibility of our experiments, we included this subset of the r-index in the repository containing our Suffixient Array index.

8.4.2 Datasets and Patterns Extraction

In this experiment, we used the three genomic datasets from Table 1: Chromosome 19, Salmonella, and SARS-CoV-2. Since our `opt-sA` implementation is designed for DNA alphabets of four characters, we preprocess the datasets to remove all non-DNA characters. This ensures that all four competing methods use the same three input texts. For each dataset, we generated three sets of 100,000 patterns of lengths 10, 100, and 1000 each. Specifically, given a text $T[1, n]$ and a pattern length m, we selected patterns by drawing uniform random positions $i \in [1, n - m + 1]$.

8.4.3 Results

The plots below show a comparison between the query times of the four indexes and the RAM throughput of our workstation when extracting substrings of length $m = 10, 100$, and 1000 from uniformly-chosen positions in a uniform text of 1 billion characters. On our workstation, we obtained the following RAM throughput benchmarks: 6.23341 ns/character for $m = 10$, 2.31762 ns/character for $m = 100$, and 1.41736 ns/character for $m = 1000$. The RAM throughput for reading the entire text of 1 billion characters was of 1.17591 ns/character. Note that the RAM throughput indicates a hard lower bound for processing a pattern of a certain length since it provides the maximum rate at which a block of length m can be retrieved from our system's internal memory. In turn, this is a hard lower bound for any deterministic pattern matching algorithm guaranteeing a correct answer.

In all experiments, the `lz77` oracle was always orders of magnitude slower than the `rlz` oracle, while always using essentially the same space. Symmetrically, the `rlz` oracle was always as fast as the `bitpacked` text, while using orders of magnitude less space. Since `rlz` always dominated the other two oracles, in Fig. 7 we only show results using the `rlz` oracle.

As the plots show, the optimization opt-sA only slightly increases the space usage of sA while dramatically speeding up query times (even by one order of magnitude

[11] https://github.com/nicolaprezza/r-index

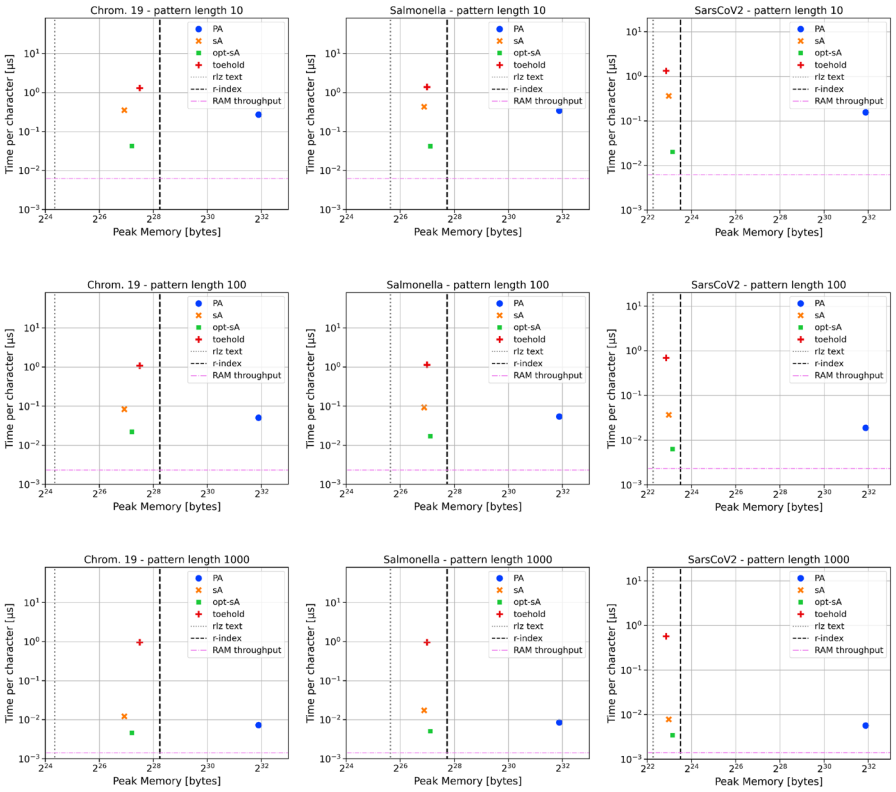

Fig. 7 Summary of results for the three biological datasets. We compare the three implementations of our index, using the RLZ-based random access oracle, against the toehold lemma of the r-index. The y-axis represents the time per character required by each index to locate one occurrence across 100,000 patterns of different lengths. The x-axis shows the peak memory recorded during each execution. Both axes are in log scale. Additionally, we include three dashed lines indicating the size of the random-access text oracle (`rlz`), the size of the original r-index, and the RAM throughput of our workstation when extracting substrings of length 10, 100, 1000 from uniform positions in a uniform text of 1 billion characters

in some cases). Our results show that opt-sA always dominates the Prefix Array by a wide margin: opt-sA was always faster than the Prefix Array while using orders of magnitude less space. As expected from our analysis of opt-sA in the I/O model (Theorem 19), opt-sA dominates the toehold lemma of the r-index by a wide margin on the moderately-repetitive Chr 19 dataset. On this dataset, opt-sA was always smaller than the toehold lemma and about *half the size* of the full r-index. At the same time, opt-sA was always *from one to two orders of magnitude faster* than the toehold lemma. On the more repetitive Salmonella and SarsCov2 datasets, opt-sA was only slightly larger than the toehold lemma and always smaller than the full r-index while at the same time being always from one to two orders of magnitude faster.

The plots show that opt-sA was always at most 1 order of magnitude slower than the RAM throughput. This is the case of Chrom. 19 and patterns of length 100. On the other hand, for patterns of length 10 and 1000 the results were much more favorable. In the best case, for SarsCoV2 and length 1000, opt-sA (about 3.5 ns per character)

was only 2.5 times slower than the RAM throughput (about 1.4 ns per character). This aligns with our analysis in the I/O model, as for long patterns and repetitive collections made up of variations of a nearly-uniform string, fewer binary search steps are expected. We remark that our implementation is not yet heavily optimized and we expect further improvements with more optimization work.

9 Future Work

We list some open problems on which we are currently working.

Queries in $O(\chi)$ Space
We have already mentioned the problem of determining whether χ is a reachable measure, i.e., $O(\chi)$ words are sufficient to store the text and to perform efficient random access. A possible way to attack this problem is to observe that our random access data structure based on string attractors (using $O(\chi \log(n/\chi))$ space) does not exploit any structural property of suffixient sets other than them being a string attractor. Suffixient sets are, however, very particular string attractors. Can we use other combinatorial properties of suffixient sets to support random access in $o(\chi \log(n/\chi))$ space?

Another interesting problem is to support counting and locating all pattern occurrences (rather than just one) in space $O(\chi)$ on top of the random access oracle. Eventually, we would also like to compute the BWT range of the pattern with our mechanism, therefore completely replacing the Burrows-Wheeler transform. Locating could be achieved, for example, by encoding the ϕ function of the r-index [18] in $O(\chi)$ space. As we briefly mentioned in the paper, this is possible in space $O(\bar{r})$ on top of the random access oracle (while retaining I/O efficiency), but $\bar{r}$ can be asymptotically larger than χ.

The Repetitiveness Measure χ
The relation $\chi \leq \bar{r}$ was always true in our experiments, even though our theory only predicts $\chi \leq 2\bar{r}$ (Lemma 10). Can we prove better upper-bounds for χ as a function of $\bar{r}$ or, more in general, as a function of other repetitiveness measures [3]?

Recently, [50] has shown that $\chi \leq 2r$ where r is the number of equal-letter runs in the BWT of the text. Since it is known that $\bar{r} = O(r \lg^2 n)$ [51], it would be interesting to see if there is such a relationship between the size of a smallest suffixient set of a string and that of the reversed.

Necessary Samples
While Suffixient Arrays are the optimal solution to the problem of minimizing the cardinality of a suffixient set (Definition 9), they are not necessarily the smallest sampling of the Prefix Array guaranteeing the correctness of Algorithm 1. In a sense, Suffixient Arrays guarantee a *sufficient* but not *necessary* condition for Algorithm 1 to work correctly. While we have determined what this necessary condition is, efficient algorithms to optimize it are still under investigation.

Optimizations

Our implementation is not yet heavily optimized: by using more cache-efficient data structures (in particular, Elias-Fano dictionaries), we believe we can get even closer to the RAM throughput of our machine. We will explore this direction in a future practical implementation of our index in the context of a usable DNA aligner for repetitive collections.

Approximate Pattern Matching

We are interested in whether suffixient sets can be adapted for use with the technique of finding all substrings of T within edit distance k of P by cutting P into $k+1$ pieces and, for each piece, trying to match that piece exactly and then match the other pieces allowing for a total of k edits to them [52]. More generally, this technique is referred to as search schemes [53]. If we can overcome the challenges then we believe suffixient sets are well-suited to this task since, if P is reasonably long and k is reasonably small, after we have exactly matched one piece we will be fairly deep in the suffix tree of T, where edge labels tend to be long and it is advantageous to descend edges without checking all the characters in their labels (which we do via LCP queries between suffixes of P and T).

The problem we face is determining all the ways to approximately match the other pieces. This can be done with backtracking in the suffix tree, but with our current implementation we do not see how to perform this backtracking: when we compute the LCP of a suffix of P and any suffix of T, we may descend several edges at once without realizing it and, for backtracking, later we should go back and partly explore the subtrees hanging off the path we descended. We are currently looking for ways to detect when we descend past a node in the suffix tree and find its string depth, so we can return to it later and visit its other children.

Acknowledgements We thank Ragnar Groot Koerkamp and Giulio Ermanno Pibiri for fruitful discussions on the topic.

Author Contributions D.C., L.D., T.G., S.K., G.M., F.O., N.P. contributed equally to the paper.

Funding *Davide Cenzato, Sung-Hwan Kim, Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.
Lore Depuydt: Funded by a PhD Fellowship FR (1117322N), Research Foundation – Flanders (FWO).
Travis Gagie: Funded by NSERC Discovery Grant RGPIN-07185-2020.
Giovanni Manzini: Funded by the NextGenerationEU programme PNRR ECS00000017 Tuscany Health Ecosystem (CUP: I53C22000780001) and by the project PAN-HUB, Italian Ministry of Health (CUP: I53C22001300001).
Francisco Olivares: Funded by scholarship ANID-Subdirección de Capital Humano/Doctorado Nacional/2021-21210579, ANID, Chile and by Basal Funds FB0001, ANID, Chile.

Data Availability We provide all the necessary information to access the datasets used in our experimental analysis in the GitHub repository of our project.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Cenzato, D., Olivares, F., Prezza, N.: On computing the smallest suffixient set. In: Proceedings of the 31st International Symposium on String Processing and Information Retrieval, pp. 73–87 (2024). https://doi.org/10.1007/978-3-031-72200-4_6
2. Depuydt, L., Gagie, T., Langmead, B., Manzini, G., Prezza, N.: Suffixient Sets. arxiv:2312.01359 (2024)
3. Navarro, G.: Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Comput. Surv. **54**(2), 29–12931 (2022). https://doi.org/10.1145/3434399
4. Navarro, G.: Indexing highly repetitive string collections, Part II: Compressed indexes. ACM Comput. Surv. **54**(2), 26–12631 (2022). https://doi.org/10.1145/3432999
5. DNA Sequencing Costs: Data. https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data. Accessed: (07 Jan 2025)
6. Medini, D., Donati, C., Tettelin, H., Masignani, V., Rappuoli, R.: The microbial pan-genome. Curr. Opin. Genetics Dev. **15**(6), 589–594 (2005). https://doi.org/10.1016/j.gde.2005.09.006
7. 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
8. Altshuler, D., Pollara, V.J., Cowles, C.R., Etten, W.J.V., Baldwin, J., Linton, L., Lander, E.S.: An SNP map of the human genome generated by reduced representation shotgun sequencing. Nature **407**, 513–516 (2000). https://doi.org/10.1038/35035083
9. Consortium, T.G.P.: A global reference for human genetic variation. Nature **526**(7571), 68–74 (2015). https://doi.org/10.1038/nature15393
10. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 319–327. ACM (1990)
11. Baeza-Yates, R.A., Gonnet, G.H.: A new approach to text searching. In: Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR '89, pp. 168–175. ACM, New York, NY, USA (1989). https://doi.org/10.1145/75334.75352
12. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: Pat trees and pat arrays. Inf. Retrieval: Data Struc. Algorithms **66**, 82 (1992)
13. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
14. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE **88**(11), 1722–1732 (2000). https://doi.org/10.1109/5.892708
15. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 390–398 (2000). https://doi.org/10.1109/SFCS.2000.892127
16. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing (STOC), pp. 397–406 (2000). https://doi.org/10.1145/335305.33535
17. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. Comput. **12**(1), 40–66 (2005)
18. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM. **67**(1) (2020). https://doi.org/10.1145/3375890
19. Puglisi, S.J., Zhukova, B.: Smaller RLZ-compressed suffix arrays. In: 2021 Data Compression Conference (DCC), pp. 213–222. IEEE (2021). https://doi.org/10.1109/DCC50243.2021.00029

20. Depuydt, L., Renders, L., Vyver, S., Veys, L., Gagie, T., Fostier, J.: b-move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index. In: Pissis, S.P., Sung, W.-K. (eds.) 24th International Workshop on Algorithms in Bioinformatics (WABI 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 312, pp. 10–11018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.WABI.2024.10
21. Li, H.: Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arxiv:1303.3997 (2013)
22. Bannai, H., Gagie, T., I., T.: Refining r-index. Theoret. Comput. Sci. **812**, 92–108 (2020). https://doi.org/10.1016/j.tcs.2019.08.005
23. Rossi, M., Oliva, M., Langmead, B., Gagie, T., Boucher, C.: MONI: A pangenomic index for finding maximal exact matches. J. Comput. Biol. **29**(2), 169–187 (2022). https://doi.org/10.1089/CMB.2021.0290
24. Boucher, C., Gagie, T., I, T., Köppl, D., Langmead, B., Manzini, G., Navarro, G., Pacheco, A., Rossi, M.: PHONI: Streamed Matching Statistics with Multi-Genome References. In: Data Compression Conference (DCC), pp. 193–202 (2021). https://doi.org/10.1109/DCC50243.2021.00027
25. Nishimoto, T., Tabei, Y.: Optimal-time queries on BWT-runs compressed indexes. In: 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021), pp. 101–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, (2021). https://doi.org/10.4230/LIPIcs.ICALP.2021.101
26. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM J. Comput. **44**, 513–539 (2015). https://doi.org/10.1137/130936889
27. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoret. Comput. Sci. **483**, 115–133 (2013). https://doi.org/10.1016/j.tcs.2012.02.006
28. Belazzougui, D., Cáceres, M., Gagie, T., Gawrychowski, P., Kärkkäinen, J., Navarro, G., Ordóñez, A., Puglisi, S.J., Tabei, Y.: Block trees. J. Comput. Syst. Sci. **117**, 1–22 (2021). https://doi.org/10.1016/j.jcss.2020.11.002
29. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: International Symposium on String Processing and Information Retrieval, pp. 201–206. Springer, (2010). https://doi.org/10.1007/978-3-642-16321-0_20
30. Bentley, J.W., Gibney, D., Thankachan, S.V.: On the complexity of BWT-runs minimization via alphabet reordering. In: Proceedings of the 28th Annual European Symposium on Algorithms (ESA 2020). LIPIcs, vol. 173, pp. 15–11513 (2020). https://doi.org/10.4230/LIPICS.ESA.2020.15
31. Kempa, D., Prezza, N.: At the roots of dictionary compression: string attractors. In: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2018). STOC 2018, pp. 827–840. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3188745.3188814 . https://doi.org/10.1145/3188745.3188814
32. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 785–794 (2009). https://dl.acm.org/doi/10.5555/1496770.1496856
33. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big BWTs. Algorithms Mol. Biol. **14**(1), 13–11315 (2019). https://doi.org/10.1186/S13015-019-0148-5
34. Li, H.: Exploring single-sample snp and indel calling with whole-genome de novo assembly. Bioinformatics **28**(14), 1838–1844 (2012). https://doi.org/10.1093/bioinformatics/bts280
35. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
36. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13
37. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm, Digital Equipment Corporation (1994). (124 Technical Report)
38. Bentley, J.W., Gibney, D., Thankachan, S.V.: On the Complexity of BWT-Runs Minimization via Alphabet Reordering. In: 28th Annual European Symposium on Algorithms (ESA 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, (2020). https://doi.org/10.4230/LIPIcs.ESA.2020.15
39. Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-length compressed indexes are superior for highly repetitive sequence collections. In: Amir, A., Turpin, A., Moffat, A. (eds.) String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November

10-12, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5280, pp. 164–175. Springer, (2008). https://doi.org/10.1007/978-3-540-89097-3_17
40. Boldi, P., Vigna, S.: Kings, Name Days, Lazy Servants and Magic. In: Ito, H., Leonardi, S., Pagli, L., Prencipe, G. (eds.) 9th International Conference on Fun with Algorithms (FUN 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 100, pp. 10–11013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.FUN.2018.10
41. Baláž, A., Gagie, T., Goga, A., Heumos, S., Navarro, G., Petescia, A., Sirén, J.: Wheeler maps. In: Soto, J.A., Wiese, A. (eds.) LATIN 2024: Theoretical Informatics, pp. 178–192. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-55598-5_12
42. Bille, P., Gørtz, I.L., Cording, P.H., Sach, B., Vildhøj, H.W., Vind, S.: Fingerprints in compressed strings. J. Comput. Syst. Sci. **86**, 171–180 (2017). https://doi.org/10.1016/j.jcss.2017.01.002
43. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692
44. Li, H.: Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics **28**(14), 1838–1844 (2012). https://doi.org/10.1093/bioinformatics/bts280
45. Gagie, T.: How to find long maximal exact matches and ignore short ones. In: International Conference on Developments in Language Theory, pp. 131–140 (2024). https://doi.org/10.1007/978-3-031-66159-4_10
46. Kuhnle, A., Mun, T., Boucher, C., Gagie, T., Langmead, B., Manzini, G.: Efficient construction of a complete index for pan-genomics read alignment. J. Comput. Biol. **27**(4), 500–513 (2020). https://doi.org/10.1089/CMB.2019.0309
47. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoret. Comput. Sci. **410**(51), 5354–5364 (2009)
48. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms **14**(3), 344–370 (1993). https://doi.org/10.1006/jagm.1993.1018
49. Belazzougui, D., Navarro, G.: Optimal lower and upper bounds for representing sequences. ACM Trans. Algorithms **11**(4), 31–13121 (2015). https://doi.org/10.1145/2629339
50. Navarro, G., Romana, G., Urbina, C.: Smallest suffixient sets as a repetitiveness measure. In: Proceedings of the 32nd International Symposium on String Processing and Information Retrieval, pp. 217–232 (2025). https://doi.org/10.1007/978-3-032-05228-5_18
51. Kempa, D., Kociumaka, T.: Resolution of the Burrows-Wheeler transform conjecture. Commun. ACM **65**(6), 91–98 (2022). https://doi.org/10.1145/3531445
52. Lam, T.W., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.: High throughput short read alignment via bi-directional BWT. In: IEEE International Conference on Bioinformatics and Biomedicine, pp. 31–36 (2009). https://doi.org/10.1109/BIBM.2009.42
53. Kucherov, G., Salikhov, K., Tsur, D.: Approximate String Matching Using a Bidirectional Index. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) Combinatorial Pattern Matching, pp. 222–231. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07566-2_23

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Davide Cenzato[1] · Lore Depuydt[2] · Travis Gagie[3] · Sung-Hwan Kim[1,4] · Giovanni Manzini[5] · Francisco Olivares[6,7] · Nicola Prezza[1]

✉ Davide Cenzato
davide.cenzato@unive.it

Lore Depuydt
lore.depuydt@ugent.be

Travis Gagie
travis.gagie@dal.ca

Sung-Hwan Kim
sunghwan.kim@unive.it; sunghwan.kim@santannapisa.it

Giovanni Manzini
giovanni.manzini@unipi.it

Francisco Olivares
folivares@uchile.cl

Nicola Prezza
nicola.prezza@unive.it

1 DAIS, Ca' Foscari University of Venice, Venice, Italy

2 Department of Information Technology - IDLab, Ghent University - imec, Ghent, Belgium

3 Faculty of Computer Science, Dalhousie University, 6050 University Ave., Halifax B3H 1W5, Nova Scotia, Canada

4 Sant'Anna School of Advanced Studies, Pisa, Italy

5 Computer Science Dept., University of Pisa, Pisa, Italy

6 Department of Computer Science, University of Chile, Santiago, Chile

7 CeBiB — Centre for Biotechnology and Bioengineering, Santiago, Chile