An LMS-based Grammar Self-index with Local Consistency Properties [⋆]

Diego Díaz-Domínguez, Gonzalo Navarro, and Alejandro Pacheco

[1] Department of Computer Science, University of Chile
{ddiaz,gnavarro,apacheco}@dcc.uchile.cl
[2] CeBiB — Center for Biotechnology and Bioengineering

Abstract. A grammar self-index of a text T (Claude et al. 2012) consists of a grammar $\mathcal{G}$ that only produces T and a geometric data structure that indexes the string cuts of the right-hand sides of $\mathcal{G}$'s rules. This representation uses space proportional to G, the size of the grammar, which is small when the text is repetitive. However, the index is slow for matching long patterns; it finds the *occ* occurrences of a pattern $P[1..m]$ in $O((m^2 + occ)\log G)$ time. The most expensive part is a set of binary searches for the different cuts $P[1..j]P[j+1..m]$ in the geometric data structure. Christiansen et al. 2010 solved this problem by building a locally consistent grammar that only searches for $O(\log m)$ cuts of P. Their representation, however, requires significant extra space (tough still in $O(G)$) to store a set of permutations for the nonterminal symbols. In this work, we propose another locally consistent grammar that builds on the idea of LMS substrings (Nong et al. 2009). Our grammar also requires to try $O(\log m)$ cuts when searching for P, but it does not need to store permutations. As a result, we obtain a self-index that searches in time $O((m\log m + occ)\log G)$ and is of practical size. Our experiments showed that our index is faster than previous grammar-based indexes at the price of increasing the space by a 1.8x factor on average. Other experimental results showed that our data structure becomes convenient when the patterns to search for are long.

Keywords: Grammar compression · LMS-substrings · Locally consistent parsing

1 Introduction

Self-indexes built on dictionary compression [19] have gained increasing attention in recent years as they can reduce the space usage of massive repetitive text collections by orders of magnitude, while still supporting direct access and pattern searches on the text. Among those, grammar-based self-indexes [13] are promising as they allow to access the text with a logarithmic-time penalty [1].

Claude and Navarro [6] proposed the first self-index based on grammar compression. Its most recent version [8] offers relevant worst-case guarantees and is

[⋆] Funded in part by Basal Funds FB0001, Fondecyt Grant 1-200038, Ph.D. Scholarship 21171332, ANID, Chile.

competitive in both time and space. Given any context-free grammar of size G representing a text $T[1..n]$, the index uses $3+\epsilon$ words of space (i.e., $(3+\epsilon)\log n$ bits) for any constant $\epsilon > 0$, and finds all the *occ* occurrences of any pattern $P[1..m]$ in time $O((m^2 + occ)\log G)$.

While this time complexity is practical for short patterns, the quadratic time complexity becomes noticeable on longer ones. There have been several attempts to decrease the m^2 term in the time complexity. In the same article [8], they show that the time can be reduced to $O(m^2 + (m+occ)\log^\epsilon G)$, for any constant $\epsilon > 0$, while maintaining the space within $O(G)$, by replacing binary searches with Patricia trees and using larger geometric data structures. More radically, Christiansen et al. [5, App. A] showed that the time complexity can be reduced to $O(m\log n + occ \log^\epsilon n)$, still within $O(G)$ space, by using new data structures called Z-fast tries instead of Patricia trees. Those improved time complexities pay a significant price, however, in the constant hidden in the $O(G)$ space term. If implemented, this index is likely to be considerably larger than the classic grammar-based index of Claude et al. [8].

The main contribution of Christiansen et al., however, is the design of a particular grammar with local consistency properties, meaning that identical text substrings are largely parsed in the same way. This feature helps to reduce the number of anchor points of P that must be tested in the text (the meaning of this concept will be made clear throughout the paper), from $m-1$ to $O(\log m)$. Such a significant reduction enables them to search in near-optimal time $O(m + (1+occ)\log^\epsilon n)$. This scheme, however, only works on the particular grammar they designed. Further, it is also likely that, if implemented, the heavy theoretical machinery required to achieve their result produces a large index in practice.

Our contribution We design a practical scheme to generate a locally consistent grammar. We encode the result using the representation of Claude et al. 2021 to produce a self-index that locates the occurrences of a pattern $P[1,m]$ in $O((m\log m + occ)\log G)$ time. Our method builds on a new idea to produce a locally consistent parsing that uses induced suffix sorting [21]. We prove this parsing is locally consistent and exploit the fact that, unlike Christiansen et al. [5], it does not require us to store the alphabet permutations of the parsing levels to perform pattern matching. When querying P in the index, we use the lexicographical relations of its symbols to infer how its occurrences in T would be parsed. By not generating the random permutations, we are likely to obtain a larger grammar than Christiansen et al. We weigh this disadvantage by simplifying our grammar in a way that does not affect its locally consistent properties. Using induced suffix sorting for grammar compression is not new, nor is the idea of simplifying the grammar [22,9]. However, self-indexing on those grammar compressors, exploiting their local consistency properties, is a contribution of this paper. Our experimental results showed that our grammar is comparable in size with that of Christiansen et al., but, as explained before, does not require the permutations. Further experiments also showed that our self-index is larger than that of Claude et al. 2021 (which we built on top of RePair), but considerably faster as the pattern length grows.

2 Related Concepts

2.1 Grammar compression

A *context-free grammar*, or just grammar, is a tuple $\mathcal{G} = (V, \Sigma, \mathcal{R}, S)$ that describes rewriting rules to produce a set of strings in Σ^*. In this tuple, V is the alphabet of *nonterminal* symbols, Σ is the alphabet of *terminal* symbols, $\mathcal{R}$ is a list of productions that maps nonterminals to strings over $\Sigma \cup V$, and $S \in V$ is the start symbol of $\mathcal{G}$. The nonterminals rewrite as strings, while terminal symbols cannot be replaced. The rules in $\mathcal{R}$ are represented as $A \to B$, where $A \in V$ and $B \in (V \cup \Sigma)^*$, meaning that A is replaced by B. The set of strings in Σ^* we can obtain from S by recursively rewriting nonterminals is the language generated by $\mathcal{G}$, $\mathcal{L}(\mathcal{G})$. The *parse tree* of a string $T \in \mathcal{L}(\mathcal{G})$ is a labeled ordinal tree that represents the recursive nonterminal replacements leading to T. The root is labeled with S, the leaves are labeled with terminals spelling out T left to right, and the internal nodes are labeled with nonterminals: the children of A are, left to right, the symbols of B for some rule $A \to B \in \mathcal{R}$.

The aim in *grammar compression* is to encode an input string $T[1..n]$ by finding a small grammar $\mathcal{G}$ whose language is $\mathcal{L}(\mathcal{G}) = \{T\}$. In this grammar there is exactly one rule $A \to B$ per $A \in V$; we call $exp(A) \in \Sigma^*$ the only string of terminals derived from A, and then $T = exp(S)$. The *size* $G = |\mathcal{G}|$ of the grammar is the sum of the lengths of all the right-hand sides of the rules. Then we significantly compress T if we manage to build a grammar of size $G \ll |T|$ that generates only T. Even approximating the smallest grammar for T within a small constant factor is NP-hard [25,4]. However, there are good heuristic that perform well in practice, RePair [16] being the most popular one.

2.2 A grammar self-index

A classical grammar self-index [7,8] for a string $T[1..n]$ consists of a grammar $\mathcal{G}$ generating (only) T and a geometric data structure [3] used to perform efficient pattern matching on T. Using $O(G)$ space, the index locates all the *occ* occurrences of a pattern $P[1..m]$ in time $O((m^2 + occ) \log G)$.

In order to use $\mathcal{G}$, it is first (easily) modified to enforce some properties:

1. For every terminal $a \in \Sigma$, there is a nonterminal rule $X_a \to a$.
2. There are no rules in $\mathcal{R}$ of the form $A \to \varepsilon$ or $A \to B$ with $B \in V$.
3. The nonterminal symbols are numbered such that, if $X < Y$, then the reverse of $exp(X)$ is lexicographically smaller than the reverse of $exp(Y)$.
4. Every nonterminal $A \in V$ appears at least twice on the right-hand sides of $\mathcal{R}$. The only exceptions are S and the nonterminals produced from Σ.

The index stores the *grammar tree* of $\mathcal{G}$ [7] (also called partial parse tree [25]), which is is a pruned version of the parse tree: if a nonterminal $A \in V$ labels several parse tree nodes, we maintain only the leftmost as an internal node and convert the others to leaves. The leaves of the grammar tree induce a partition of T into *phrases*, formed by the substrings $exp(A)$ for the labels A of all the leaves. Those phrases are indexed in the geometric data structure.

When searching for $P[1..m]$ in the grammar self-index, we classify its occurrences into *primary* and *secondary.* Primary occurrences of P cross two or more phrases, while secondary occurrences are completely contained within phrases.

The pattern matching algorithm first reports the primary occurrences of P, using the geometric structure to find all the distinct sequences of consecutive phrases that contain P as a substring. For each such sequence, it obtains the lowest common ancestor v of their corresponding grammar tree leaves. Say the label of v is $A \in V$. We traverse upwards from A to the root S to find the position of the occurrence in T (for every A child of A' in the grammar tree, the index stores the offset of $exp(A)$ inside $exp(A')$). Let $A = A_0$, A_1, A_2, $\ldots$ be the labels of the nodes traversed in the way to S. Apart from reporting the primary occurrence, the algorithm also finds all the leaves in the grammar tree labeled A_i and reports further (secondary) occurrences of P inside $exp(A_i)$. The ancestors of those leaves labeled A_i recursively trigger further secondary occurrences. The total time amortizes to constant per occurrence thanks to the grammar transformation rules applied.

The grid data structure We use the geometric data structure to locate the primary occurrences of P. To build it, we first define two string sets; the first one, $\mathcal{Y}$, has $|\mathcal{R}|$ strings, and the second, $\mathcal{X}$, has $G - |\mathcal{R}| + |\Sigma|$ strings. The sets are built as follows; let $A \to B_1 \ldots B_t \in \mathcal{R}$ be any nonterminal rule and let v the internal node for A in the grammar tree. For every B_i, with $i \in [1..t]$, we insert the reverse sequence $exp(B_i)^r$ to $\mathcal{Y}$. Additionally, for every proper suffix $B_i \ldots B_t$, with $i \in [2, t]$, we insert the string $exp(B_i) \cdots exp(B_t)$ to $\mathcal{X}$. We build a matrix M of $|\mathcal{Y}| \times |\mathcal{X}|$ cells. Every row k is labeled with the string in $\mathcal{Y}$ with lexicographic rank k. Equivalently, every column k' is labeled with the string in $\mathcal{X}$ with lexicographic rank k'. The cell of M in the intersection of the row for $exp(B_i)^r$ and the column for $exp(B_{i+1}) \cdots exp(B_t)$ stores the identifier of the $(i+1)$th child of v in the grammar tree. This arrangement automatically gives the lowest common ancestor v of the primary occurrence and the offset inside it.

Finding the primary occurrences We cut P in two halves $P[1..j]$ and $P[j+1..m]$. The idea is to locate the range of rows (y_1, y_2) in M whose labels are prefixed by $P[1..j]^r$ and the range (x_1, x_2) of columns prefixed by $P[j+1..m]$. The non-empty cells within the grid range (x_1, y_1, x_2, y_2) point to the internal nodes in the grammar tree with primary occurrences of P. We binary search for $P[1..j]^r$ in the prefixes of $\mathcal{Y}$ to define (y_1, y_2), and binary search for $P[j+1..m]$ in the prefixes of $\mathcal{X}$ to define (x_1, x_2). When comparing $P[1..j]^r$ against the row labels, we decompress from the grammar tree the last j characters of the reversed string in $\mathcal{Y}$. Similarly, when comparing $P[j+1..m]$ against the column labels, we decompress the first $m-j+2$ symbols of the string in $\mathcal{X}$. The time for both searches is then $O(m \log G)$ per partition of P, adding up to $O(m^2 \log G)$ in total. We can reduced this to $O(m^2)$, still within $O(|G|)$ space, by using Patricia trees on $\mathcal{Y}$ and $\mathcal{X}$. Once we obtain a grid range, we get the p points inside it in time $O((1+p)\log G)$, which over all the partitions adds up to $O((m^2 + occ)\log G)$.

Christiansen et al. [5, App. A] showed that one can obtain time $O(m \log n + occ \log^{\epsilon} n)$ for any constant $\epsilon > 0$ within space $O(G)$, but the constant multiplying their space is much higher and the resulting index is likely impractical.

2.3 Locally consistent parsing

Locally consistent parsing [26,18] is a method for partitioning a text $T[1..n]$ into a sequence of *phrases* in which equal substrings of T are largely parsed in the same way. Let a *phrase boundary* be a pair of positions $(j-1, j)$ such that $T[j]$ is a prefix in the kth phrase and $T[j-1]$ is a suffix in the phrase preceding it. A parsing is locally consistent if there are two integers a, b (which may depend on n) such that, for every pair of equal substrings $T[j..j+u] = T[j'..j'+u]$, only their first a and their last b phrase boundaries can differ. In general, a locally consistent parsing algorithm puts a phrase boundary in T wherever some specific symbol combination arises. The first and last phrases of $T[j..j+u]$ might be formed in a different way than those in $T[j'..j'+u]$ because they might be preceded or followed by different symbols. Note that this approach differs from other parsing algorithms such as Lempel-Ziv [27] or RePair [16], which use global information on T to define its partition.

2.4 Locally consistent grammars

This type of grammars is constructed by applying successive rounds of locally consistent parsing over $T[1..n]$. In every round i, we capture the distinct phrases in the input text T^i ($T^1 = T$) and create new nonterminals rewriting to them. We then build a new text T^{i+1} by replacing the phrases in T^i with their corresponding nonterminal symbols. This new text T^{i+1} is the input for the next round. The algorithm stops when T^i can no longer be partitioned. If the phrases in every T^i are of length at least 2, then the string T^{i+1} is at most half the length of T^i, and thus the number of parsing rounds is $O(\log n)$ and the total running time is of the same order as for parsing T.

The algorithm described above produces a balanced grammar $\mathcal{G}$, which is probably bigger than the one we obtain with RePair. In exchange, if a pattern P appears more than once in T, then the parse subtrees containing its occurrences will be almost identical, differing only in a few nodes at the ends of every tree level. The internal part of the subtrees remains unchanged regardless P's context. This can be exploited to speed up pattern matching.

Recently, Christiansen et al. [5] proposed a run-length locally-consistent grammar of size $G = O(\gamma \log(n/\gamma))$, where γ is the size of the smallest attractor of

T [12]. In their algorithm, the parsing rounds have two steps. In the first one, they create new nonterminal rules with the equal-symbol runs of T^i. These rules are of the form $X \rightarrow x^l$, where x^l is a run of l copies of symbol x in T^i (these rules are of constant size). Then, they produce a new string $\hat{T}^i$ by replacing the runs with their generating nonterminals. In the second round step, they define a random permutation $\pi : \hat{\Sigma}^i \rightarrow [1..|\hat{\Sigma}^i|]$ for the symbols in the alphabet $\hat{\Sigma}^i$ of $\hat{T}^i$, and use this permutation to partition $\hat{T}^i$: each phrase ends in a local minima, which is a position $\hat{T}^i[j]$ such that $\pi(\hat{T}^i[j-1]) > \pi(\hat{T}^i[j]) < \pi(\hat{T}^i[j+1])$.

Christiansen et al. also showed that, if we build the self-index of Claude et al. [8] using their grammar, then we require to test only $O(\log m)$ cuts of P to find its primary occurrences in $\mathcal{G}$. Their idea consists in preprocessing P at query time with the same algorithm they used to build $\mathcal{G}$. In every round i, they obtain the symbols of $\hat{P}^i$ by querying the equal-symbol runs of P^i in a hash table storing the nonterminals assigned to the sequences of the form x^l in $\mathcal{G}$. Subsequently, they query the phrases induced by the local minima of $\hat{P}^i$ in another hash table that maps phrases (right-hand sides in $\mathcal{R}$) to their nonterminal symbols. They use the symbols returned from the lookups to compute P^{i+1}. The prefix $\hat{P}^i[1..a]$ and the suffix $\hat{P}^i[b..|P^i|]$ that are not complete phrases do not have symbols in P^{i+1}. Analogously, the first and last equal-symbol runs of P^i do not have symbols in $\hat{P}^i$ as they are (possibly) incomplete. The preprocessing yields a list Q with the positions in P that limit incomplete parsing phrases. More specifically, every position $q \in Q$ is either the rightmost symbol under the internal node for $\hat{P}^i[a]$ in P's parse tree, or the leftmost symbol under the node for $\hat{P}^i[b]$. The elements in Q denote the cuts we try in the geometric data structure. As there are $O(1)$ incomplete phrases per parsing level i, there are $O(\log m)$ cuts in total. The time obtained [5] is $O(m + (occ + 1)\log^{\epsilon} n)$ for any constant $\epsilon > 0$.

2.5 Induced suffix sorting

Induced suffix sorting (ISS) [21] is a technique to sort the suffixes of T in lexicographical order. The basic idea consists of sampling some suffixes, sort them in lexicographical order, and use the result to induce the orders of the rest. ISS is the underlying procedure in several linear-time algorithms that build the suffix array [21,20,17] and the Burrows-Wheeler transform (BWT) [23,2]. We repeat some ISS-related definitions introduced for the SA-IS algorithm [21]:

Definition 1. *Let $T[1..n]$ be a text terminated with a sentinel symbol \$, smaller than the others. A position $T[i]$ is called L-type if $T[i] > T[i+1]$ or if $T[i] = T[i+1]$ and $T[i+1]$ also L-type. Instead, $T[i]$ is said to be S-type if $T[i] < T[i+1]$ or if $T[i] = T[i+1]$ and $T[i+1]$ is also S-type. The symbol $T[n] = \$$ is S-type.*

Definition 2. *A character $T[i]$ is called leftmost S-type, or LMS-type, if $T[i]$ is S-type and $i = 1$ or $T[i-1]$ is L-type.*

Definition 3. *An LMS substring is (i) a minimal substring $T[i..j]$ with both $T[i]$ and $T[j]$ being LMS characters, for $i < j$; or (ii) the sentinel itself.*

Let $D[1..n]$ be an array that stores in $D[i]$ the type (L, S, or LMS) of $T[i]$. We refer to D as the *description* of T.

SA-IS is a recursive approach. In every recursion level i, it first scans the input text T^i, with $T^1 = T$, from right to left to compute its description. As it moves through the text, SA-IS records the positions of the LMS substrings and sorts them. The idea is to use the resulting ranks to induce an ordering for the suffixes of T that are not prefixed by LMS substrings. When comparing two LMS substrings $T[a..b]$ and $T[a'..b']$, the algorithm inspects them from left to right until their symbols differ. However, if for some two positions $j \in [a..b]$ and $j' \in [a'..b']$, the type of $T[j]$ is distinct from the type of $T[j']$, then the substring with S-type gets the highest order, even if $T[j]$ and $T[j']$ are the same symbol. We refer to this ordering as $\prec_{LMS}$.

Before inducing the order of the suffixes, SA-IS recursively sorts the suffixes starting LMS substrings. For this purpose, it creates a new string T^{i+1} in which it replaces the LMS substrings with their $\prec_{LMS}$ orders, and uses T^{i+1} as input for another recursive call, of level $i+1$. The recursive call returns the suffix array of T^{i+1}, which gives the order between LMS-starting suffixes. This information is used to induce the order of the other suffixes of T^i.

Nunes et al. [22] noticed that the LMS substrings of the recursive calls of SA-IS can be used to build a grammar for T. The advantage of this construction is that it requires $O(n)$ time and space, and it is much cheaper in practice than other popular grammar heuristics, like RePair. However, it does not achieve compression ratios as good as those of RePair.

3 A Grammar Self-Index based on LMS Parsing

3.1 LMS parsing

We define *LMS parsing* as the procedure of parsing T using its LMS substrings. The idea is similar to the method described in the SA-IS algorithm: we compute the description of T, and define a phrase $T[i..i']$ for every consecutive pair of LMS-type positions $T[i-1]$ and $T[i']$. We refer those phrases as *LMS phrases*.

The LMS parsing is locally consistent. To prove it, we demonstrate that equal substrings of T have the same descriptions, except possibly at their endpoints.

Lemma 1. *The LMS parsing is locally consistent.*

Proof. Let $T[a..b] = T[a'..b']$ be two equal substrings. Let their suffixes of length $u \geq 1$ be equal-symbol runs, and symbols $T[b-u]$ and $T[b'-u]$ be different from $T[b-u+1]$ and $T[b'-u+1]$, respectively. The symbols within the same run have the same types, by definition. However, those types might differ if $T[b]$ and $T[b']$ are followed by different symbols. In particular, if $T[b-u+1]$ is L-type and $T[b'-u+1]$ is not, then $T[b'-u+1]$ can be LMS-type, and thus a phrase may end at $T[b'-u+1]$ and not at $T[b-u+1]$ (or vice versa).

Instead, the positions $T[b-u]$ and $T[b'-u]$ preceding those runs will always have the same type because they are followed by the same symbol, $T[b-u+$

$1] = T[b' - u + 1]$. Furthermore, the equal substrings $T[a + 1..b - u - 1]$ and $T[a' + 1..b' - u - 1]$ will also have the same types because they are preceded and followed by the same symbols.

Finally, the types of $T[a]$ and $T[a']$ may differ because they may depend on the preceding symbol. Both or none can be L-type, but if they are not, then one may be LMS-type and the other be S-type, depending on the symbols at $T[a-1]$ and $T[a' - 1]$. Therefore, one substring may have an LMS phrase ending at the first position and not the other.

To conclude, there can be at most one LMS phrase boundary appearing in each extreme of one of the substrings and not in the order. $\square$

We then produce a locally consistent grammar $\mathcal{G}$ using several rounds of LMS parsing. In every round i, we create a dictionary $\mathcal{D}^i$ with all the distinct LMS phrases of T^i. Then, for every $F \in \mathcal{D}^i$, we create a new rule $X \rightarrow F$, where X is the number of rules in $\mathcal{R}$ built before round i plus the $\prec_{LMS}$ rank of F among the strings in $\mathcal{D}^i$. After generating the new rules, we create T^{i+1} by replacing the LMS phrases in T^i with their nonterminal symbols. If there are still repeated symbols in T^{i+1}, we perform another parsing round $i + 1$ using T^{i+1} as input.

Note that this procedure is very similar to that of Christiansen et al. [5]. They randomly permute the alphabet and place a phrase boundary after every local minimum. In our LMS parsing, we place a phrase boundary after every LMS-type symbol, which is also a local minimum. The key difference is that Christiansen et al. need to store the permutations used in order to replicate the same process on the search pattern, whereas our parsing is given by the lexicographic order and thus can be applied on the pattern without further information.

To further reduce the grammar, we create a new rule $Y \rightarrow X^l$ for every maximal equal-symbol run X^l appearing on a right-hand side. The grammar tree represents rules $Y \rightarrow X^l$ as $Y \rightarrow X\,X^{l-1}$, where X^{l-1} is a special leaf. This unique cut is enough to detect the occurrences of any pattern, provided a special procedure is carried out to report the secondary occurrences inside X^{l-1} [5].

We also reduce space by replacing the nonterminals appearing once with the right-hand sides of their rules, unless they represent equal-symbol runs. The rules of those replaced symbols are then removed from $\mathcal{R}$.

3.2 Computing the cuts during the pattern matching

We use our grammar to build the self-index of Claude et al. [8] (with the special provision for run-length rules). The main change is the way we cut the pattern.

Let us call the *projection* of $P^i[p]$ the index $q \in [1..m]$ such that $P[p]$ is the rightmost leaf under the subtree rooted at $P^i[p]$ in P's parse tree; similarly with the projection of $\hat{P^i}[p]$.

Our procedure for finding the cuts for P is analogous to that of Christiansen et al. [5]. We start with an empty set Q and apply successive rounds of LMS parsing over P. In every round i, we insert into Q the projection of $P^i[1]$ and $\hat{P^i}[1]$. We then hash the distinct LMS phrases in P^i. We discard for the hashing, however, the prefix $P^i[1..a]$ where $P^i[a]$ is the leftmost LMS-type symbol, and the suffix $P^i[b..]$ where $P^i[b-1]$ is the rightmost LMS-type symbol. These elements can be incomplete phrases, so we do not use them for the next round. Still, we do record in Q the projection of $P^i[a]$ and the projection of $P^i[b-1]$ in P's parse tree. Additionally, when the rightmost equal-symbol run of P^i (last symbol in $\hat{P^i}$) has length $u > 1$ and $P^i[|P^i| - u]$ is L-type, we also insert the projection of $P^i[|P^i| - u + 1]$ into Q. We consider this position because if the rightmost run of P^i is S-type, then $P^i[|P^i| - u + 1]$ can be LMS-type if $P^i[|P^i| - u]$ is L-type. After scanning P^i, we sort the hashed phrases in $\prec_{LMS}$ order and create a new string P^{i+1} that replaces the phrases' occurrences with their $\prec_{LMS}$ orders. The new parse P^{i+1} is the input for the next round. The processing of P stops when there are no more LMS-type symbols in P^i.

The length of P^{i+1} is at most half that of P^i, so we scan $O(m)$ symbols along the $O(\log m)$ parsing rounds. On the other hand, we sort the distinct LMS subtrings of P^i in $O(|P^i|)$ time [21], which adds up to $O(m)$ total time. Therefore, the complete preprocessing of P requires $O(m)$ time.

To find the primary occurrences, we binary search the cut $P[1..q]P[q+1..m]$ associated to every $q \in Q$. Since $|Q| \in O(\log m)$, the total time to look for the primary occurrences is $O(|Q|m \log G) \subseteq O(m \log m \log G)$, plus $O(|Q| \log G) \subseteq O(\log m \log G)$ for the geometric searches, plus $O(occ \log G)$ to extract the grid points. Our final result borrows the space figures of Claude et al. [8].

Theorem 1. *Let our grammar, built for $T[1..n]$, be of size G. Then our index uses $G \log n + (2+\epsilon)G \log G$ bits of space, for any constant $\epsilon > 0$, and finds all the occ occurrences of $P[1..m]$ in time $O((m \log m + occ) \log G)$.*

We note that, still within $O(G)$ space, we can use Patricia trees to speed up the binary searches, obtaining $O(m \log m + (\log m + occ) \log G)$ time.

4 Experiments

We implemented our version of the grammar index in `C++` on top of the `SDSL-lite` library [11]. The source code is available at `https://github.com/ddiazdom/LPG/tree/LPG_grid`. We generated two versions of our index. The regular version (**lms-ind**) implements the wavelet tree of the grid data structure using plain bit vectors. The second variant (**lms-ind-rrr**) encodes the wavelet tree using the RRR [24] data structure for compressed bit vectors. We compared our software against the state-of-the-art self-indexes for repetitive collections:

- **r-ind**[3]: The run-length compressed FM-index of [10]. It uses $O(r \log n)$ bits, where r is the number of runs in the text's BWT, and supports locate within that space.
- **lz-ind**[4]: A self-index based on Lempel-Ziv [15] that guarantees $O(z \log z)$ bits of space over a Lempel-Ziv parse of z phrases. We also included the variant that uses LZ-end parsing (**lz-end-ind**) [14].

[3] https://github.com/nicolaprezza/r-index
[4] https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/LZ

Dataset	n	σ	RePair	LMS	LMS post	LC
para	429,265,758	5	5,344,480	22,787,047	8,933,303	8,888,002
cere	461,286,644	5	4,069,450	37,426,507	6,802,801	4,069,450
influenza	154,808,555	15	1,957,370	4,259,746	3,304,035	4,477,322
einstein.en	467,626,544	139	212,903	643,338	427,142	601,755
kernel	257,961,616	162	1,374,650	3,769,839	2,870,350	3,795,801

Table 1. Datasets. The second and third columns are the number of symbols and alphabet, respectively. The rest of the columns are grammar sizes (value for G) obtained with different grammar algorithms. The column for RePair considers the postprocessing described in Claude et al. 2021 (Sec. 3). LMS is the grammar obtained with LMS parsing and LMS post is grammar resulted from the postprocessing described in Section 3. The last column (LC) refers to the grammar of Christiansen et al. 2020.

- **slp-ind** : An optimized implementation of the grammar index of Claude et al. 2012 [7]. This version speeds up the binary searches by storing q-grams of the prefixes to which the nonterminals expand (grid labels). We used three q-gram values in our experiments; 4,8, and 16. We refer to these variants as **slp-ind4**, **slp-ind8** and **slp-ind16**, respectively.
- **g-ind**[5]: The grammar index of Claude et al. 2021. The first variant of this index (**g-ind-bs**) uses binary searches over the grid labels to find the primary occurrences of P. The second variant (**g-ind-pt**) speeds up the search by maintaining two Patricia trees, one for a subset of uniformly sampled column labels in the grid and the other for a subset of uniformly sampled row labels. We used three sampling rates; 1/4, 1/16, and 1/64. We refer to these variants as **g-ind-pt4**, **g-ind-pt16**, and **gt-ind-pt64**, respectively.

We used five data sets of the *Pizza&Chilli*[6] corpus for the experiments. The characteristics of these datasets are shown in Table 1. We assessed the compression ratio and the running time for locating patterns. We extracted random substrings from the datasets and then we searched them back with the different indexes. The length of these patterns ranged from 10 to 100 characters.

We also compared our grammar algorithm against RePair and the method of Christiansen et al. [5]. The metric we used for the comparison was the grammar size (G value). The algorithm of Christiansen et al. has no formal implementation, so we produced one ourselves.

All the experiments were carried out on a machine with eight Intel(R) Xeon(R) CPU E5-2407 processors at 2.40 GHz and 250 GB RAM. We compiled our source code using full compiler optimizations and we do not use multi threading.

[5] https://github.com/apachecom/grammar_improved_index

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

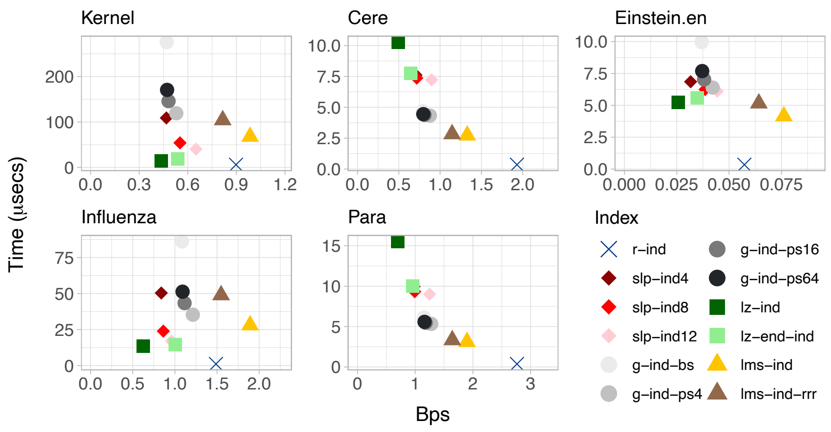

Fig. 1. Time-space tradeoffs for locating 1000 random pattern of length 100 on different collections and indexes. The time (y-axis) is given in μsecs per occurrence and the index space (x-axis) in bits per symbol (bps).

5 Results and Discussion

The grammars produced with our method were, on average, 4.2 times bigger than the RePair grammars (see columns 4 and 5 of Table 1). This considerable difference is expected as our grammar algorithm prioritizes consistency over compression. By further processing the grammars produced with our method (see Section 3), we reduced their sizes by 41% on average. However, their final sizes were still far from those of RePair; they were 1.82 times bigger on average. Interestingly, the sizes of our post-processed grammars were similar to those of Christiansen et al. (columns 6 and 7 of Table 1), even though we are not using random permutations. It is important to note that the grammar of Christiansen et al. cannot be further simplified without losing local consistency.

Figure 1 shows the trade-offs between index space usage and time for locating patterns of length 100. The results varied widely depending on the dataset. For instance, in *cere* and *para*, the index that used the most space was **r-ind** (1.93 and 2.76 bps, respectively), but it was also the fastest (0.34 and 0.37 μsecs). The second-largest index was **lms-ind** (1.32 and 1.89 bps), and the second-fastest after **r-ind**. In both datasets, the variant **lms-ind-rrr** reduced the space usage and stayed competitive for locating, but in the other datasets, **lms-ind-rrr** reduced the space at the cost of becoming slower. The smallest representation in *cere* and *para* was **lz-ind** (0.49 and 0.70 bps), but it was the slowest at locating (10.2 and 15.5 μsecs). In *einstein.en*, **lms-ind** was the biggest index (0.076 bps), even bigger than **r-ind**, which remained the fastest. However, **lms-ind** was the fastest dictionary-based data structure. The **lms-ind-rrr** variant reduced the space, but it did not outperform **r-ind**.

Things went differently with *influenza* and *kernel*. The Lempel-Ziv data structures (**lz-ind** and **lz-end-ind**) were competitive with **r-ind** for locating

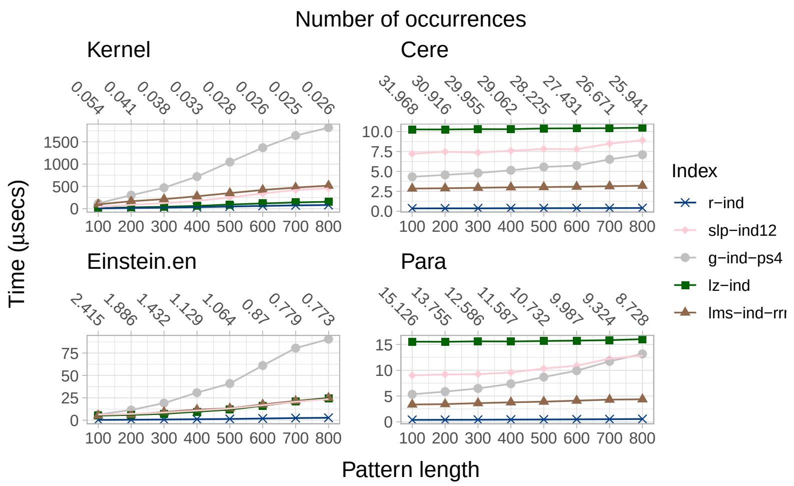

Fig. 2. Locating time for increasing pattern lengths. The time is given in μsecs per occurrence. The upper x-axes indicate the number of occurrences (in millions) searched in every combination of document and pattern length.

(14.16 and 18.05 μsecs versus 5.89 μsecs, respectively). Nevertheless, they used less space. This result is unexpected as *influenza* and *kernel* are not as repetitive as *einstein.en*. Our index performed poorly in these datasets. Both variants (**lms-ind** and **lms-ind-rrr**) were the biggest dictionary-based data structures, and they were not the fastest ones. However, they were competitive with **r-ind** in terms of space, with **lms-ind-rrr** using less space than **r-ind** in *kernel* (0.81 bps versus 0.89 bps, respectively), although they were significantly slower.

Figure 2 shows the performance of the indexes for the locate operation using different pattern lengths (from 100 to 800). In *para* and *cere*, **lms-ind** greatly outperformed the other dictionary-based indexes as the pattern length increased. This was not the case in *einstein.en* and *kernel*, where the performance of **lms-ind-rrr** was not different from that of **slp-ind**. We also noted that in those datasets, the performance of **lz-ind** was very close to that of **r-ind**. Interestingly, the performance of **r-ind** remained steady as the pattern length increased.

6 Concluding remarks

We presented a locally consistent grammar that allows us to produce a self-index using $G \log n + (2 + \epsilon)\ G \log G$ bits of space and that performs pattern matching in $O((m \log m + occ) \log G)$ time. Our experimental results showed that our method is a practical alternative to the technique of Christiansen et al. as we obtain a locally consistent grammar of comparable size without storing the symbol permutations. The resulting self-index is thus not much larger than the other popular dictionary-based indexes but generally faster at locating patterns, especially long ones.

References

1. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random Access to Grammar-Compressed Strings and Trees. SIAM Journal on Computing **44**(3), 513–539 (2015)
2. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big BWTs. Algorithms for Molecular Biology **14**(1), Article 13 (2019)
3. Chan, T., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. 27th Annual Symposium on Computational Geometry (SoCG). pp. 1–10 (2011)
4. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory **51**(7), 2554–2576 (2005)
5. Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-Time Dictionary-Compressed Indexes. ACM Transactions on Algorithms **17**(1), Article 8 (2020)
6. Claude, F., Navarro, G.: Self-Indexed Grammar-Based Compression. Fundamenta Informaticae **111**(3), 313–337 (2011)
7. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 180–192 (2012)
8. Claude, F., Navarro, G., Pacheco, A.: Grammar-compressed indexes with logarithmic search time. Journal of Computer and System Sciences **118**, 53–74 (2021)
9. Díaz-Domínguez, D., Navarro, G.: A grammar compressor for collections of reads with applications to the construction of the BWT. In: Proc. 31st Data Compression Conference (DCC) (2021)
10. Gagie, T., Navarro, G., Prezza, N.: Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. Journal of the ACM **67**(1), 1–54 (2020)
11. Gog, S., Beller, T., Moffat, A., Petri, M.: From Theory to Practice: Plug and Play with Succinct Data Structures. In: Proc. 13th International Symposium on Experimental Algorithms (SEA). pp. 326–337 (2014)
12. Kempa, D., Prezza, N.: At the roots of dictionary compression: string attractors. In: Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC). pp. 827–840 (2018)
13. Kieffer, J.C., Yang, E.H.: Grammar-based codes: a new class of universal lossless source codes. IEEE Transactions on Information Theory **46**(3), 737–754 (2000)
14. Kreft, S., Navarro, G.: LZ77-Like Compression with Fast Random Access. In: Proc. 10th Data Compression Conference (DCC). p. 239–248 (2010)
15. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science **483**, 115–133 (2013)
16. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE **88**(11), 1722–1732 (2000)
17. Louza, F., Gog, S., Telles, G.P.: Inducing enhanced suffix arrays for string collections. Theoretical Computer Science **678**(1), 22–39 (2017)
18. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)
19. Navarro, G.: Indexing Highly Repetitive String Collections , Part II : Compressed Indexes. ACM Computing Surveys **54**(2), Article 26 (2021)

20. Nong, G.: Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Transactions on Information Systems **31**(3), 1–15 (2013)
21. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. 19th Data Compression Conference (DCC). pp. 193–202 (2009)
22. Nunes, D.S.N., Louza, F.A., Gog, S., Ayala-Rincón, M., Navarro, G.: A grammar compression algorithm based on induced suffix sorting. In: Proc. 28th Data Compression Conference (DCC). pp. 42–51 (2018)
23. Okanohara, D., Sadakane, K.: A linear-time Burrows-Wheeler transform using induced sorting. In: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 90–101 (2009)
24. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4), Article 43 (2007)
25. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science **302**(1-3), 211–222 (2003)
26. Sahinalp, C., Vishkin, U.: Data compression using locally consistent parsing. Tech. rep., UMIACS Technical Report (1995)
27. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on information theory **23**(3), 337–343 (1977)