Grammar-Compressed Indexes with Logarithmic Search Time [⋆]

Francisco Claude[1], Gonzalo Navarro[2,3], and Alejandro Pacheco[3]

[1] LinkedIn, USA
[2] Center for Biotechnology and Bioengineering (CeBiB), Chile
[3] Department of Computer Science, University of Chile

Abstract. Let a text $T[1..n]$ be the only string generated by a context-free grammar with g (terminal and nonterminal) symbols, and of size G (measured as the sum of the lengths of the right-hand sides of the rules). Such a grammar, called a *grammar-compressed representation* of T, can be encoded using $G \lg G$ bits. We introduce the first grammar-compressed *index* that uses $O(G \lg n)$ bits (precisely, $G \lg n + (2+\epsilon) G \lg g$ for any constant $\epsilon > 0$) and can find the *occ* occurrences of patterns $P[1..m]$ in time $O((m^2 + occ) \lg G)$. We implement the index and demonstrate its practicality in comparison with the state of the art, on highly repetitive text collections.

1 Introduction and Related Work

Grammar-based compression is an active area of research since at least the seventies [CRA76,Sto77,ZL78,SS82]. A given sequence $T[1..n]$ over alphabet $[1..\sigma]$ is replaced by a hopefully small (context-free) grammar $\mathcal{G}$ that generates exactly one string: T. Let g be the number of grammar symbols, counting terminals and nonterminals. Let $G = |\mathcal{G}|$ be the *size* of the grammar, measured as the sum of the lengths of the right-hand sides of the rules. Then a basic grammar-compressed representation of T requires $G \lg G$ bits, instead of the $n \lg \sigma$ bits required by a plain representation. It always holds $G \geq \lg n$, and indeed G can be as small as $O(\lg n)$ in extreme cases; consider $T = a^n$.

Grammar-based methods yield universal lossless source codes[1] if one ensures to avoid some obvious redundancies and properly encodes the grammar [KY00]. On the other hand, unlike statistical methods, which exploit frequencies to achieve compression, grammar-based methods exploit repetitions in the text; this makes them especially suitable for compressing highly repetitive sequence collections, where statistical compression is helpless [KN13,Nav20]. Highly repetitive collections, containing long identical substrings that are possibly far away from each other, arise when managing software repositories, versioned documents, transaction logs, periodic publications, and computational biology sequence databases, among others.

Finding the smallest grammar $\mathcal{G}^*$ that represents a given text T is NP-complete [Sto77,SS82,Ryt03,CLL+05]. Moreover, the size G^* of the smallest grammar is never smaller than the number z of phrases in a Lempel-Ziv parse [LZ76] of T. A simple method to achieve an $O(\lg n)$-approximation to the smallest grammar size is to parse T using Lempel-Ziv and then convert the parse into a grammar [Ryt03,CLL+05]. More precisely, these and other approximations

⋆ A preliminary version of this article appeared in *Proc. SPIRE'12* [CN12]. This work was partially funded by Basal Funds FB0001, Fondecyt Grant 1-200038, and Doctoral Scholarship grant 21180760, ANID, Chile.

1 That is, its size asymptotically converges to the entropy of any finite-state information source over a fixed alphabet.

[Ryt03,CLL$^+$05,Jez15,Jez16] yield grammars of size $G = O(z \lg(n/z)) \subseteq O(G^* \lg(n/G^*))$, that is, with an approximation ratio of $O(\lg(n/G^*))$. It has also been shown that no approximation ratio better than $\Omega(\lg n / \lg\lg n)$ to z is possible in general [HLR16].

The known approximation ratios of popular grammar compressors such as LZ78 [ZL78], Re-Pair [LM00] and Sequitur [NMWM94], instead, are much larger than the optimal [CLL$^+$05,HLR16,BHH$^+$19]. Still, some of those methods (in particular Re-Pair) perform very well in practice, both in classical and repetitive settings.[2]

On the other hand, unlike Lempel-Ziv, grammar compression allows one to decompress arbitrary substrings of T in logarithmic time [GKPS05,BLR$^+$15,BCPT15]. The most recent results extract any $T[p..p+\ell-1]$ in time $O(\ell+\lg n)$ [BLR$^+$15] and even $O(\ell/\lg_\sigma n+\lg n)$ [BCPT15], which is close to optimal [VY13]. Unfortunately, those representations require $O(G \lg n)$ bits, possibly proportional but in practice many times the size of the output of a grammar compressor.

More ambitious than just extracting substrings from T is to ask for *indexed searches*, that is, finding the *occ* occurrences in T of a given pattern $P[1..m]$. *Self-indexes* are compressed text representations that support both operations, *extract* $T[p..p+\ell-1]$ and *locate* the occurrences of a pattern $P[1..m]$, in time sublinear (and usually polylogarithmic) in n. They appeared in the year 2000 and have focused mostly on statistical compression [NM07]. As a result, they work well on classical texts, but not on repetitive collections [MNSV10,Nav20]. Some of those self-indexes have been adapted to such repetitive collections [MNSV10,NPL$^+$13,NPC$^+$13,DJSS14,BGG$^+$14,BCG$^+$15,GNP20], but they do not reach the compression ratio of the best grammar-based methods.

Searching for patterns on grammar-compressed text has been faced mostly in sequential form [AB92], that is, scanning the whole grammar. The best result [KMS$^+$03] achieves time $O(G+m^2+occ)$. This may be $o(n)$, but is still linear in the size of the compressed text. There exist a few self-indexes based on LZ78-like compression [FM05,RO08,ANS12], but LZ78 is among the weakest grammar-based compressors. In particular, LZ78 has been shown not to be competitive on highly repetitive collections [MNSV10].

The only self-index supporting general grammar compressors [CN10] operates on "binary grammars", where the right-hand sides of the rules are of length 1 or 2. Given such a grammar they achieve, among other tradeoffs, $g \lg n + (3+o(1)) g \lg g$ bits of space and $O(m(m+h) \lg g \lg\lg g + occ \cdot h \lg g)$ search time, where $h \le g$ is the height of the parse tree of the grammar. A general grammar of g symbols and size G can be converted into a binary grammar by adding at most $G-2g$ symbols and/or rules.

A self-index based on Lempel-Ziv compression was also developed [KN13]. It uses $z \lg z + 2z \lg n + O(z \lg \sigma)$ bits of space and searches in time $O(m^2 \bar{h} + (m+occ) \lg z)$, where $\bar{h} \le z$ is the nesting of the parsing. Extraction requires $O(\ell\, \bar{h})$ time. Experiments on repetitive collections [CFMPN10,CFMPN16] showed that the grammar-based compressor [CN10] can outperform the (by then) best classical self-index adapted to repetitive collections [MNSV10] but, at least that particular implementation, was not competitive with the Lempel-Ziv-based self-index [KN13].

The search times in both self-indexes depend on h or $\bar{h}$. This is undesirable as both are only bounded by g or z, respectively. As mentioned, this kind of dependence has been removed for extracting text substrings [BLR$^+$15], at the cost of using $O(G \lg n)$ further bits. Similarly, a recent result [GJL19] shows that from any grammar of size G one can obtain a balanced grammar of size

[2] See the statistics in `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

$O(G)$ generating the same string; therefore one can replace h by $\lg n$ in previous results at the cost of raising the space to $O(G \lg n)$. Previous work [CN10] then achieves $O(m(m+\lg n)\lg G + occ \lg n \lg G)$ time within $O(G \lg n)$ bits of space.

There have also been combinations of grammar-based and Lempel-Ziv-based methods [GGK$^+$12,GGK$^+$14,BEGV18,CE18], yet (1) none of those is implemented, (2) the constant factors multiplying their space complexities are usually large, (3) they cannot be built on a given arbitrary grammar. They use at least $O(z \lg(n/z) \lg n)$ bits (which is an *upper bound* to our space complexity) and can search as fast as in $O(m + \lg^\epsilon z + occ(\lg^\epsilon z + \lg\lg n))$ time for any constant $\epsilon > 0$ [CE18], decreasing to $O(m + occ \lg\lg n)$ time with $O(z \lg(g/z) \lg\lg z \lg n)$ bits of space [BEGV18]. Gagie et al. [GGK$^+$12] can depart from any given grammar, but add some extra space so that, within $O(G \lg n + z \lg\lg z \lg n)$ bits, they can search in time $O(m^2 + (m + occ) \lg\lg n)$.

Gagie et al. [GNP20] built an index based on a completely different measure: the number r of equal-letter runs in the Burrows-Wheeler Transform (BWT) [BW94] of the text. This measure is sensitive to repetitiveness, but usually larger than z or G. In exchange, the index is very fast: $O((m + occ) \lg\lg n)$ search time, which also shows up in their implementation.

More recently, Navarro and Prezza [NP19] introduced a self-index of $O(\gamma \lg(n/\gamma) \lg n)$ bits, where $\gamma \le z \le G$ is the size of any *attractor* of T (which lower-bounds many other repetitiveness measures). Kociumaka et al. [KNP20] derived another self-index of $O(\delta \lg(n/\delta) \lg n)$ bits, where $\delta \le \gamma$ is an even stricter measure of repetitiveness. Both can search in time $O(m \lg n + occ \lg^\epsilon n)$. Kociumaka et al. [CEK$^+$20] maintain the space of the first index [NP19] while improving its time to $O(m + (occ + 1) \lg^\epsilon n)$, and reach optimal time $O(m + occ)$ within $O(\gamma \lg(n/\gamma) \lg^{1+\epsilon} n)$ bits of space. All these results are theoretically appealing, but still suffer from the drawbacks (1)–(3) above, and still their size dominate only an upper bound on $G \lg n$.

In this article we introduce the first (as of the time of conference publication [CN12], and still the only[3]) grammar-based self-index that can be built from *any given grammar* of size G, using $G \lg n + (2 + \epsilon) G \lg G$ bits for any $\epsilon > 0$, and whose search time depends only logarithmically on G, independently of the grammar height. In addition, we give an engineered implementation of the index and compare it with different state-of-the-art indexes on repetitive collections, showing that our index is also practical. In fact, the ability of our index to build on any grammar has an important practical value, because it can be built on top of compressors like RePair, which perform extremely well in practice.

The following theorem summarizes its properties; we note that the space can be simplified, as in the abstract, to $G \lg n + (2 + \epsilon') G \lg g$ for any constant $\epsilon' > \epsilon$, and the search time can be simplified to $O((m^2 + occ) \lg G)$ because $\lg\lg n \le \lg G$. Table 1 compares our result with previous work.

Theorem 1. *Let a sequence $T[1..n]$ be represented by a context-free grammar with g symbols, size G and height h. Then, for any $0 < \epsilon \le 1$, there exists a data structure using at most $G \lg n + 2G \lg g + \epsilon\, g \lg g + o(G \lg g) + O(G)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O((m^2/\epsilon) \lg\lg n + (m + occ)(1/\epsilon + \lg g / \lg\lg g))$. It can extract any substring of length ℓ from T in time $O(\ell + h \lg(G/h))$. The structure can be built in $O(G \lg G \lg n)$ time and $O(G \lg^2 n)$ bits of working space.*

[3] Kociumaka et al. [CEK$^+$20, App. A] show how to build an index of $O(G \lg n)$ bits on top of any grammar, which searches in time $O(m \lg n + occ \lg^\epsilon n)$. This is still theoretical work and is preceded by the present article.

Table 1. Our result in the context of other implemented indexes based on measures z (number of Lempel-Ziv phrases), r (number of BWT runs), G (grammar size) and g (grammar rules). Our space is simplified and assumes ϵ is a constant; our construction time assumes the grammar is given (many can be built in $O(n)$ time).

Source	Technique	Space in bits	Query time	Construction time
[CN10]	Binary grammars	$g \lg n + 3g \lg g + o(g \lg g)$	$O(m(m+h) \lg g \lg \lg g + occ \cdot h \lg g)$	$O(n + g \lg n)$
[KN13]	Lempel-Ziv	$2z \lg n + z \lg z + O(z \lg \sigma)$	$O(m^2 \bar{h} + (m + occ) \lg z)$	$O(n \lg \sigma)$
[GNP20]	BWT runs	$O(r \lg n)$	$O((m + occ) \lg \lg n)$	$O(n)$
Ours	Grammars	$G \lg n + (2+\epsilon) G \lg G$	$O((m^2 + occ) \lg G)$	$O(G \lg^2 n)$

Note that the extraction time still depends on the grammar height. To remove this dependence, we can include the structure of Belazzougui et al. [BCPT15], which adds $O(G \lg n)$ bits. Within that space we derive a coarser version of our result.

Corollary 1. *Let a sequence $T[1..n]$ over alphabet $[1..\sigma]$ be represented by a context-free grammar with g symbols and of size G. Then there exists an index requiring $O(G \lg n)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O(m^2 + (m + occ) \lg^{\epsilon} g)$, for any $\epsilon > 0$, and extracts any substring of length ℓ from T in time $O(\ell / \lg_{\sigma} n + \lg n)$.*

As most of the previous work, our results holds on the so-called transdichotomous RAM model, where the computer word holds $w = \Theta(\lg n)$ bits and can perform all the usual arithmetic and logical operations (including multiplication) in constant time.

In the rest of the article we describe our structure. First, we preprocess the grammar to enforce several invariants useful to ensure our time complexities. Then we use a data structure for binary relations [BCN13] to find the "primary" occurrences of P, that is, those formed when concatenating symbols in the right hand side of a rule. To get rid of the factor h in this part of the search, we extend a technique [GKPS05] to extract the first m symbols of the expansion of any nonterminal in time $O(m)$. To find the "secondary" occurrences (i.e., those that are found as the result of the nonterminal containing primary occurrences being mentioned elsewhere), we use a pruned representation of the parse tree of T. This tree is traversed upwards for each secondary occurrence to report. The grammar invariants introduced ensure that those traversals amortize to a constant number of steps per occurrence reported. In this way we get rid of the factor h on the secondary occurrences too.

We also show that our structure is practical. We implement the index of Theorem 1 and show that it outperforms the preceding grammar-based index [CN10], even in its optimized form [CFMPN16], and it becomes a valid space/time tradeoff to the Lempel-Ziv based self-index [KN13] (also in optimized form [CFMPN16]). Our expermental results show that, while the technique to speed up the extraction [GKPS05] does not have an impact in practice, the idea to amortize the cost of finding the secondary occurrences does speed up the index significantly.

The main differences with our conference version [CN12] are improved theoretical complexities, the whole implementation and experimental results, and an expanded and improved writing.

2 Basic Concepts

2.1 Sequence Representations

Our data structures use succinct representations of sequences. Given a sequence $S[1..N]$, over the alphabet Σ, we need to support the following operations:

- $\mathit{access}(S, i)$ retrieves the symbol $S[i]$;
- $\mathit{rank}_a(S, i)$ counts the number of occurrences of a in $S[1..i]$;
- $\mathit{select}_a(S, j)$ computes the position where the jth a appears in S.

For the case $|\Sigma| = 2$ (i.e., bitmaps), all the operations can be supported in $N + o(N)$ bits and constant time [Cla96]. Sadakane and Okanohara [OS07] proposed a compressed representation based on Elias-Fano codes [Eli74,Fan71], which is useful when the number N_1 of 1s in S is small. It takes $N_1 \lg \frac{N}{N_1} + O(N_1)$ bits of space and supports $\mathit{select}_1(S, j)$ in constant time and $\mathit{access}(S, i)$ and $\mathit{rank}(S, i)$ in time $O(\lg \min(N_1, N/N_1))$.

For general sequences, we will use a representation [BN15] that requires $N \lg |\Sigma| + o(N \lg |\Sigma|)$ bits and solves $\mathit{access}(S, i)$ in $O(1)$ time and $\mathit{select}(S, j)$ and $\mathit{rank}(S, i)$ in time $O(\lg \lg_w |\Sigma|)$.

2.2 Labeled Binary Relations

A labeled binary relation is a binary relation $\mathcal{R} \subseteq A \times B$, where $A = [1..n_1]$ and $B = [1..n_2]$, augmented with a function $\mathcal{L} : A \times B \to L \cup \{\perp\}$, $L = [1..\ell]$, that defines labels for each pair in $\mathcal{R}$, and $\perp$ for pairs that are not in $\mathcal{R}$. Let us identify A with the columns and B with the rows in a table. In our case, each element of A will be associated with exactly one element of B, so $|\mathcal{R}| = n_1$. We augment a representation of unlabeled binary relations [BCN13] with a plain string $S_{\mathcal{L}}[1..n_1]$ on alphabet $[1..\ell]$, where $S_{\mathcal{L}}[i]$ is the label of the pair of column i. The total space of this structure is $n_1(\lg n_2 + \lg \ell) + o(n_1 \lg n_2)$ bits. With this representation we can answer, among others, the following queries of interest in this article:

1. Find the label of the pair (a, b) associated with a given a, $S_{\mathcal{L}}[a]$, in $O(1)$ time.
2. Given a_1, a_2, b_1, and b_2, enumerate the k pairs $(a, b) \in \mathcal{R}$ such that $a_1 \leq a \leq a_2$ and $b_1 \leq b \leq b_2$, in time $O((k+1)(1+\lg n_2/\lg\lg(n_1+n_2)))$. This corresponds to operation `rel_acc`, implemented through `rel_min_obj_acc` or `rel_min_lab_acc` [BCN13, Lem. 10 or 11].

2.3 Succinct Tree Representations

There are many representations for trees $\mathcal{T}$ with N nodes that take $2N + o(N)$ bits of space. In this paper we use one called Fully-Functional (FF) [NS14], which in particular answers in constant time the following operations (node identifiers v are associated with a position in $[1..2N]$):

- $\mathit{node}(p)$ is the node with preorder number p;
- $\mathit{preorder}(v)$ is the preorder number of node v;
- $\mathit{leafrank}(v)$ is the number of leaves to the left of v;
- $\mathit{leafselect}(j)$ is the jth leaf;
- $\mathit{intrank}(v)$ is the number of internal nodes before v, in preorder;

- *intselect*(j) is the jth internal node, in preorder;
- *numleaves*(v) is the number of leaves below v;
- *parent*(v) is the parent of v;
- *child*(v, k) is the kth child of v;
- *nextsibling*(v) is the next sibling of v;
- *degree*(v) is the number of children of v;
- *depth*(v) is the depth of v; and
- *level-ancestor*(v, k) is the kth ancestor of v.

The FF representation is obtained by traversing the tree in DFS order and appending to a bitmap a 1 when we arrive at a node, and a 0 when we leave it. The operations *leafrank*, *leafselect*, *intrank*, and *intselect* are not discussed so widely in the literature. In the FF sequence $F[1..2N]$, each internal node starts with a bit 1 followed by another 1, and each leaf is represented by a 1 followed by a 0. The same mechanisms described in Section 2.1 to support *rank* and *select* for 0s and 1s on bitmaps are easily extended to support two-bit operations, within $o(N)$ extra bits. Therefore, we implement $\mathit{leafrank}(i) = \mathit{rank}_{10}(F, i-1)$, $\mathit{leafselect}(j) = \mathit{select}_{10}(F, j)$, $\mathit{intrank}(i) = \mathit{rank}_{11}(F, i-1)$, and $\mathit{intselect}(j) = \mathit{select}_{11}(F, j)$, all in constant time.

3 Preprocessing the Grammar

We will work on a given context-free grammar $\mathcal{G}$ that generates a single string $T[1..n]$ over alphabet $\Sigma = [1..\sigma]$, formed by g (terminal and nonterminal) symbols. We assume for simplicity that all the terminal symbols in Σ appear in T, otherwise we can reduce σ by renaming the nonterminals.[4] The grammar $\mathcal{G}$ then contains σ terminal symbols and $g - \sigma$ nonterminal symbols, each nonterminal X_i defined by a unique rule of the form $X_i \rightarrow \alpha_i$. The sequence α_i, called the *right-hand side* of the rule, is the sequence of terminal and non-terminal symbols to which X_i expands in one step. We call $G = \sum |\alpha_i|$ the *size* of $\mathcal{G}$. Note it holds $\sigma \le G$ since the symbols must appear in the right-hands of the rules. We assume that all the nonterminals are reachable (i.e., used to generate T); otherwise unused rules can be found and dropped in $O(G)$ time. The grammar cannot have loops since it generates a finite string T.

Let X_s always denote the start symbol (despite of successive symbol renamings). We call $\mathcal{F}(X_i)$ the single string generated by X_i, that is $\mathcal{F}(a) = a$ for terminals a and $\mathcal{F}(X_i) = \mathcal{F}(A_{i_1}) \cdots \mathcal{F}(A_{i_k})$ for nonterminals $X_i \rightarrow A_{i_1} \ldots A_{i_k}$. The grammar $\mathcal{G}$ generates the text $T = \mathcal{L}(\mathcal{G}) = \mathcal{F}(X_s)$.

For the purpose of building our index, we preprocess $\mathcal{G}$ as follows:

- First, for each terminal symbol $a \in \Sigma$ we create a rule $X_a \rightarrow a$, and replace all other occurrences of a in the grammar by X_a. As a result, the grammar contains exactly g nonterminal symbols $\mathcal{X} = \{X_1, \ldots, X_g\}$, each associated with a rule $X_i \rightarrow \alpha_i$, where $\alpha_i \in \Sigma$ or α_i is a sequence of elements in $\mathcal{X}$.
- Any rule that generates just one single nonterminal $X_i \rightarrow X_j$, or the empty string, $X_i \rightarrow \varepsilon$, is removed by replacing X_i by X_j or by ε everywhere. This decreases g without increasing G and ensures $G \ge g - \sigma$ because there are no empty right-hand sides.

[4] If needed, we can store for example a perfect hash function that maps in constant time from the symbols of Σ that appear in T to identifiers in $[1..\sigma]$, and an array mapping the symbols back to Σ. This requires $O(\sigma \lg |\Sigma|)$ further bits, which will be ignored in the sequel.

- We further preprocess $\mathcal{G}$ to enforce the property that any nonterminal X_i, except X_s and those $X_a \rightarrow a \in \Sigma$, must be mentioned in at least two right-hand sides. We traverse the rules of the grammar, count the occurrences of each symbol, and then rewrite the rules, so that only the rules of those X_i appearing more than once (or the excepted symbols) are preserved, and as we rewrite their right-hand sides, we replace any (non-excepted) X_i that appears once by its right-hand side α_i. This transformation takes $O(G)$ time and can only reduce G and g.
- Our last preprocessing step, and the most expensive one, is to renumber the nonterminals so that $i < j \Leftrightarrow \mathcal{F}(X_i)^{rev} < \mathcal{F}(X_j)^{rev}$, where "<" between strings stands for the lexicographic order and S^{rev} is string S read backwards (the purpose of this renumbering will be apparent later). The sorting can be done in $O(G \lg(n/G) + g \lg g \lg n)$ time and $O(G \lg^2 n)$ bits of space: we first compute in time $O(G \lg(n/G))$ a compressed longest-common-extension data structure [I17] on the reversed grammar and then use any sorting algorithm, where each string comparison is decided in the $O(\lg n)$ time needed to access both strings at the symbol following their longest common extension.

From now on, g will refer to the number of rules in the transformed grammar $\mathcal{G}$ (i.e., the number of terminal and nonterminal symbols in the original grammar, minus possible reductions). Instead, G will still denote the size of the original grammar (the transformed one has size at most $G + \sigma$).

Finally, we define a structure that will be key in our index (cf. pruned parse tree [Ryt03]).

Definition 1. *The* grammar tree *of $\mathcal{G}$ is a tree $\mathcal{T}_{\mathcal{G}}$ with nodes labeled in $\mathcal{X}$. Its root is labeled X_s and its topology is obtained by pruning the parse tree of T with two rules: (1) for each nonterminal symbol X_i, every node labeled X_i except the first one (in DFS order) is converted into a leaf (i.e., its subtree is pruned); (2) for each terminal a, the child a of every node labeled X_a is also pruned, leaving X_a as a leaf. We say that each X_i is* defined *in the only internal node of $\mathcal{T}_{\mathcal{G}}$ labeled X_i.*

Since each right-hand side $\alpha_i \notin \Sigma$ is written once in the tree as the sequence of children of X_i, and the root X_s is written once, the total number of nodes in $\mathcal{T}_{\mathcal{G}}$ is $G + 1$. The number of internal nodes is $g - \sigma$, and the number of leaves is $G + 1 - g + \sigma \leq G$.

Example. Figure 1 shows the reordering and grammar tree for a grammar generating the string `"alabaralalabarda"`.

The grammar tree partitions T in a way that will be useful for extracting substrings and finding occurrences.

Definition 2. *Let $X_{l_1}, X_{l_2}, \ldots$ be the nonterminals labeling the consecutive leaves of $\mathcal{T}_{\mathcal{G}}$. Let $T_i = \mathcal{F}(X_{l_i})$, then $T = T_1 \cdot T_2 \cdots T_{G+1-g+\sigma}$ is a partition of T according to the leaves of $\mathcal{T}_{\mathcal{G}}$.*

Table 2 summarizes the notation.

4 Extracting Text

We first describe a simple structure that extracts the text of a whole nonterminal, of length ℓ, in $O(\ell)$ time. We then augment this structure to support extracting any prefix or suffix of length ℓ of a nonterminal in time $O(\ell)$. Finally, we use those tools to extract any substring of T of length ℓ in

$\bar{X}_1 \to a$		$X_1 \to a$	a
$\bar{X}_2 \to b$		$X_2 \to X_9 X_1 X_6 X_9 X_5 X_1$	$alabaralalabarda$
$\bar{X}_3 \to d$		$X_3 \to b$	b
$\bar{X}_4 \to l$		$X_4 \to X_1 X_6 X_1 X_3$	$alab$
$\bar{X}_5 \to r$	$\Rightarrow$	$X_5 \to d$	d
$\bar{X}_6 \to \bar{X}_1 \bar{X}_5$		$X_6 \to l$	l
$\bar{X}_7 \to \bar{X}_1 \bar{X}_4 \bar{X}_1 \bar{X}_2$		$X_7 \to r$	r
$\bar{X}_8 \to \bar{X}_7 \bar{X}_6$		$X_8 \to X_1 X_7$	ar
$\bar{X}_9 \to \bar{X}_8 \bar{X}_1 \bar{X}_4 \bar{X}_8 \bar{X}_3 \bar{X}_1$		$X_9 \to X_4 X_8$	$alabar$

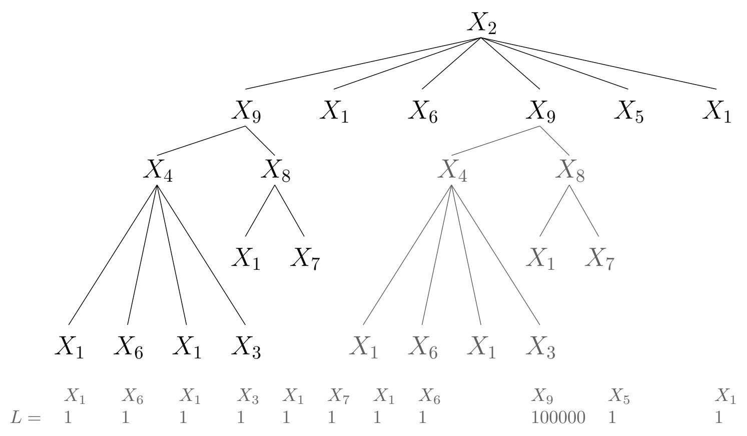


Fig. 1. At the top left, a grammar $\mathcal{G}$ generating string `"alabaralalabarda"`. At the top right, our reordering of the grammar and strings $\mathcal{F}(X_i)$. On the bottom, the grammar tree $\mathcal{T}_\mathcal{G}$ in black; the whole parse tree includes also the grayed part. Below the tree we show our bitmap L (Section 4.3).

time $O(\ell + h \lg(G/h))$. This is not the best extraction time that can be obtained [BLR$^+$15,VY13], but those more sophisticated methods require significantly more space in practice.

The optimal-time extraction of prefixes and suffixes is fundamental for supporting searches, and is obtained by extending the structure proposed by Gasieniec et al. [GKPS05] for binary grammars to general context-free grammars.

Conceptually, we represent the topology of the grammar tree $\mathcal{T}_\mathcal{G}$ using FF (Section 2.3), using $O(G)$ bits. The sequence of nonterminal labels associated with the tree nodes is stored in preorder in a sequence $X[1..G+1]$ using $G \lg g + o(G \lg g)$ bits with the representation described in Section 2.1. We also store a bitmap $C[1..g]$ that marks the rules of the form $X_i \to a \in \Sigma$ with 1s. Since the rules have been renumbered in (reverse) lexicographic order, every time we find a rule X_i such that $C[i] = 1$, we can determine the terminal symbol it represents as $a = rank_1(C, i)$ in constant time. In our example of Figure 1 this bitmap is $C = 101011100$.

This conceptual arrangement will be modified and expanded in the sequel; Table 3 gives the complete list of structures used for extracting substrings. The following theorem states our results; note that the space is the sum of those in the top part of the table after redefining ϵ appropriately.

Table 2. Notation.

Symbol	Meaning
$T[1..n]$	The text to be represented
n	The length of T
$\mathcal{G}$	A context-free grammar that generates (only) T
Σ	The alphabet of T and the set of terminals of $\mathcal{G}$
σ	Size of the alphabet $\Sigma = [1..\sigma]$
g	Number of symbols (terminals and nonterminals) of $\mathcal{G}$
G	Size of $\mathcal{G}$, that is, sum of lengths of right-hand sizes of rules
X_s	Start symbol of $\mathcal{G}$
$\mathcal{F}(X_i)$	The substring of T that symbol X_i expands to
$X_a \rightarrow a$	Introduced rules that generate terminals a; they are later renumbered to be some X_i
$X_i \rightarrow \alpha_i$	The other rules, with $\lvert\alpha_i\rvert \geq 2$
$\mathcal{T}_{\mathcal{G}}$	The grammar tree of $\mathcal{G}$
T_i	The substring of T covered by the ith leaf of $\mathcal{T}_{\mathcal{G}}$
S^{rev}	String S read backwards
$P[1..m]$	The pattern whose occurrences we find with the index
m	The length of P
$P_1 \cdot P_2$	A cut of $P = P_1 \cdot P_2$ for searching purposes

Theorem 2. *Let a sequence $T[1..n]$ be represented by a context-free grammar with g symbols and size G. Then, for any $0 < \epsilon \leq 1$, there exists a data structure using at most $G \lg g + G \lg(n/G) + (2+\epsilon) g \lg g + o(G \lg g) + O(G)$ bits of space that expands any rule of length ℓ in time $O(\ell)$, the prefix or suffix of length ℓ of any rule in time $O(\ell/\epsilon)$, and extracts any substring of length ℓ from T in time $O(\ell/\epsilon + \lg(n/G))$.*

4.1 Expanding Whole Rules

Expanding a rule X_i that does not correspond to a terminal is done as follows. By the definition of $\mathcal{T}_{\mathcal{G}}$, the first left-to-right occurrence of X_i in sequence X corresponds to the definition of X_i; all the others are leaves in $\mathcal{T}_{\mathcal{G}}$. Therefore, $v = node(select_i(X, 1))$ is the node in $\mathcal{T}_{\mathcal{G}}$ where X_i is defined. We then traverse the subtree rooted at v in DFS order. Every time we reach a leaf u, we compute its label X_j with $j = X[preorder(u)]$, and either output the terminal $rank_1(C, j)$ if $C[j] = 1$, or recursively expand X_j. This is in fact a traversal of the *parse tree* starting at node v, using the grammar tree instead. If we extract the whole sequence $\mathcal{F}(X_i)$, we perform $O(|\mathcal{F}(X_i)|)$ traversal steps, since we have removed unary paths during the preprocessing of $\mathcal{G}$ and thus the subtree rooted at v has less than $2|\mathcal{F}(X_i)|$ nodes.

Lemma 1. *Given i in $[1..g]$, we can extract $\mathcal{F}(X_i)$ in $O(|\mathcal{F}(X_i)|)$ time by using $(g-\sigma)\lceil\lg(g-\sigma)\rceil$ bits on top of $\mathcal{T}_{\mathcal{G}}$, X, and C.*

Proof. The only obstacle to having constant-time steps in the procedure we described are the queries $select_i(X, 1)$. As these are used to find the internal node of $\mathcal{T}_{\mathcal{G}}$ that defines X_i, we use a different mechanism. Note that the 0s in bitmap C correspond to the $g - \sigma$ nonterminals of the grammar (ordered by increasing nonterminal identifier). That is, if X_i is a nonterminal, then $C[i] = 0$, and

Table 3. Conceptual and actual structures used for extraction (top part) and, in addition, for searching (bottom part).

Structure	Meaning	Space in bits
$\mathcal{T}_\mathcal{G}$	Topology of the parse tree using parentheses	$O(G)$
$X[1..G+1]$	Sequence of nonterminal labels of the nodes in $\mathcal{T}_\mathcal{G}$	Not represented
$C[1..g]$	Bitmap marking which nonterminals X_i represent a symbol	$O(g)$
$\pi[1..g-\sigma]$	Permutation mapping from reverse lexicographic order of nonterminals to DFS rank of internal nodes	$(g-\sigma)\lceil\lg(g-\sigma)\rceil$
$\pi^{-1}[1..g-\sigma]$	Structure to compute inverses of π in time $O(1/\epsilon)$	$\epsilon(g-\sigma)\lg(g-\sigma)+O(g)$
$X'[1..G-g+\sigma+1]$	The elements of X corresponding to leaves	$(G-g+\sigma-1)\lg g(1+o(1))$
$\mathcal{T}_S$	Trie topology of nonterminal labels of reversed leftmost/rightmost paths in the parse tree	$O(g)$
$X_S[1..g]$	Sequence of the labels in trie $\mathcal{T}_S$ (two of them)	Not represented
$X'_S[1..g-\sigma]$	The elements of X_S that are not root children	$(g-\sigma)\lg g$
$B[1..g]$	Bitmap marking which elements of X_S are root children	$O(g)$
$X_S^{-1}[1..g]$	Inverse permutation of X_S (two of them)	$2\epsilon \cdot g\lg g$
$L[1..n]$	Compressed bitmap marking starting positions of the grammar tree leaves in T	$G\lg(n/G)+O(G)$
$\mathcal{R}$, $S_\mathcal{L}$	Binary relation connecting rule suffixes with their preceding element, $S_\mathcal{L}$ is the sequence of column labels	$(G-g+\sigma)(\lg g+\lg G)+o(G\lg g)$
P_A, P_B	Patricia trees with sampled expansions of rules	$O(G)$

$j = \mathit{rank}_0(C,i)$ gives a unique identifier for X_i in $[1..g-\sigma]$. We store a permutation $\pi[1..g-\sigma]$ where $\pi[j]=k$ if X_i labels the kth internal node of $\mathcal{T}_\mathcal{G}$, in DFS order. Using $(g-\sigma)\lceil\lg(g-\sigma)\rceil$ bits to store π in plain form, we find in constant time that X_i is defined at the node $\mathit{intselect}(\pi[\mathit{rank}_0(C,i)])$. □

The total space required considering the FF representation, sequence X, bitmap C, and permutation π, is bounded by $G\lg g + (g-\sigma)\lg(g-\sigma) + o(G\lg g) + O(G)$ bits.[5] We reduce the space by almost $(g-\sigma)\lg(g-\sigma)$ bits by removing the labels of the internal nodes, which we can recover using the inverse permutation π^{-1}.

Lemma 2. *Given i in $[1..g]$, we can extract $\mathcal{F}(X_i)$ in $O(|\mathcal{F}(X_i)|)$ time by using $G\lg g + \epsilon\, g\lg g + o(G\lg g) + O(G)$ bits, for any constant $0 < \epsilon \le 1$.*

Proof. Instead of storing the full $X[1..G+1]$, we store a reduced sequence $X'[1..G-g+\sigma+1]$ where the labels of the internal nodes are removed. We can still access any $X[p] = X'[\mathit{leafrank}(v)+1]$, with $v = \mathit{node}(p)$, if v is a leaf (because *leafrank* gives the DFS position of v skipping internal nodes). If v is an internal node, we have that $k = \mathit{intrank}(v)+1$ is its DFS position skipping leaves, so by definition of π we have that $\pi[j]=k$ if the label of v is X_i, where i is the position of the jth 0 in C, $i = \mathit{select}_0(C,j)$. Therefore,

$$X[p] \;=\; \mathit{select}_0(C, \pi^{-1}[\mathit{intrank}(v)+1]).$$

To compute π^{-1}, we use the representation of Munro et al. [MRRR12] that takes $(1+\epsilon)(g-\sigma)\lg(g-\sigma)+O(g-\sigma)$ bits and computes any $\pi[j]$ in $O(1)$ time and any $\pi^{-1}[k]$ in time $O(1/\epsilon)$. This

[5] It could be that $g = O(1)$, so $o(G\lg g)$ does not necessarily absorb $O(G)$.

yields the promised space: X would use $G \lg g$ bits and is replaced by X', which uses $(G-g+\sigma+1)\lg g$ bits, a reduction of $(g-\sigma)\lg g$ bits. In exchange, we must store $\epsilon(g-\sigma)\lg(g-\sigma)+O(g-\sigma)$ further bits to represent π^{-1}. The lemma writes the resulting space in simplified form.

Note that, with this representation, the time to access $X[i]$ is now $O(1/\epsilon)$. The extraction time stays $O(|\mathcal{F}(X_i)|)$, however, because it only accesses X at leaf nodes, which takes $O(1)$ time independently of ϵ. $\square$

With the representation of Lemma 2 we can also support general *select* on X in time $O(\lg\lg g)$: since the first occurrence of each distinct symbol X_i is removed in X' and all remaining symbols in X' are the labels of the leaves in $\mathcal{T}_\mathcal{G}$, we can compute $select_i(X,j)$ for $j>1$ by finding the position k of the $(j-1)$th occurrence of i in X', which is the leaf rank of the node in $\mathcal{T}_\mathcal{G}$. With *leafselect* we obtain its node v, and finally with *preorder* we find its DFS position, that is, the place where the symbol would occur in X:

$$select_i(X,j) \;=\; preorder(leafselect(select_i(X',j-1)));$$

while $select_i(X,1)$ is solved in $O(1)$ time with $intselect(\pi[rank_0(C,i)])$ as shown in Lemma 1.

4.2 Optimal Expansion of Rule Prefixes and Suffixes

To extract rule prefixes or suffixes of length ℓ in time $O(\ell)$, we extend the algorithm of Gasieniec et al. [GKPS05] so as to handle general grammars instead of only binary grammars. Using their notation, call $S(X_i)$ the string of labels of the nodes in the path from any node labeled X_i to its leftmost leaf in *the parse tree* (we take as leaves the nonterminals $X_a \in \mathcal{X}$ with $X_a \rightarrow a$, not the terminals $a \in \Sigma$). We insert all the strings $S(X_i)^{rev}$ into a trie (or digital tree [Fre60]) $\mathcal{T}_S$. Note that each symbol X_i labels only one node in $\mathcal{T}_S$ because, being there a single rule for each nonterminal, the sequence of leftmost descendants $S(X_i)$ is always the same for all places where X_i appears in the parse tree. Therefore, $\mathcal{T}_S$ has g nodes.

Again, we represent the topology of $\mathcal{T}_S$ using FF; the operations on this tree will use the subindex $\mathcal{T}_S$ to distinguish them from the operations in $\mathcal{T}_\mathcal{G}$. Since each X_i appears exactly once as a label, the sequence of labels $X_S[1..g]$ is a permutation of $[1..g]$. We represent it once again with the structure [MRRR12] that takes $(1+\epsilon)g\lg g$ bits and computes any $X_S[i]$ in constant time and any $X_S^{-1}[j]$ in time $O(1/\epsilon)$. To further save space, since the label of the ath root child is nonterminal $X_a \rightarrow a$, we do not store X_S but a bitmap $B_S[1..g]$ marking with 1s the preorders that are root children and a reduced version $X'_S[1..g-\sigma]$ with the labels of the other nodes. It then holds that, if $B_S[i]=0$, then $X_S[i]=X'_S[rank_0(B_S,i)]$, and otherwise $X_S[i]$ is X_a, with $a=rank_1(B_S,i)$ (we can obtain the index of X_a with $select_1(C,a)$ but we will actually want to know a in order to display it). The structure now takes $(g-\sigma)\lg g+\epsilon g\lg g+O(g)$ bits.

Lemma 3. *With structures $\mathcal{T}_S$, B_S, and X'_S, we can determine $\mathcal{F}(X_i)[1]$ in constant time given i.*

Proof. We determine the first terminal in the expansion of X_i, which labels node $v \in \mathcal{T}_S$, as follows. Since the last symbol in $S(X_i)$ is a nonterminal X_a with $X_a \rightarrow a$ for some $a \in \Sigma$ (so $\mathcal{F}(X_i)[1]=a$), it follows that X_i descends in $\mathcal{T}_S$ from X_a, which is a child of the trie root. If v is at depth d in $\mathcal{T}_S$, then this child of the root is the ancestor of v at distance $d-1$, that is,

$$v_a = level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v)-1).$$

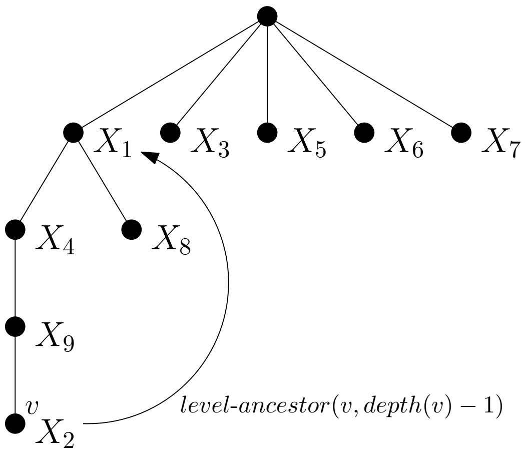


Fig. 2. Example of the trie $\mathcal{T}_S$ of leftmost paths for the grammar of Figure 1. The arrow pointing from X_2 to X_1 illustrates the procedure to determine the first terminal symbol generated by X_2.

The symbol X_a is then in X_S, at the preorder position of v_a, and then we obtain a from X_a using *rank* on the bitmap C:

$$a \;=\; rank_1(C, X_S[preorder_{\mathcal{T}_S}(v_a)]).$$

From our representation of X_S, however, we can simply compute $a = rank_1(B_S, preorder_{\mathcal{T}_S}(v_a))$. $\square$

Example. Figure 2 shows an example of this query in the trie $\mathcal{T}_S$ for the grammar presented in Figure 1. To obtain the first symbol of $\mathcal{F}(X_2)$, we map X_2 to the node $v \in \mathcal{T}_S$ and then jump to its ancestor that is a child of the root, labeled X_1. Then X_1 must represent a terminal, indeed, $C[1] = 1$ and it corresponds to the symbol $rank_1(C, 1) = 1$, which we interpret as a.

We can then extract a prefix of $\mathcal{F}(X_i)$ in optimal time.

Lemma 4. *By adding $(g - \sigma)\lg g + \epsilon g \lg g + O(g)$ bits to the data structures of Lemma 2, we can extract any $\mathcal{F}(X_i)[1..\ell]$ in time $O(\ell/\epsilon)$, given i.*

Proof. First, we obtain the corresponding node $v \in \mathcal{T}_S$ with $v = node_{\mathcal{T}_S}(X_S^{-1}[i])$. Then we obtain the leftmost symbol of v as in Lemma 3. The remaining symbols descend from the second and following children, in the parse tree, of the nodes in the upward path from a node labeled X_i to its leftmost leaf, or which is the same, in the second-to-last children of a node labeled $S(X_i)[1]$, then the second-to-last children of a node labeled $S(X_i)[2]$, and so on. These symbols $S(X_i)[j]$ label, precisely, the downward path from the root of $\mathcal{T}_S$ to v, that is, $S(X_i)[j]$ is the label of the node $level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v) - j)$. Therefore, for each node

$$u \in \langle level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v) - 2), level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v) - 3), \ldots, parent_{\mathcal{T}_S}(v), v\rangle,$$

we compute the corresponding label $X_{j'} = S(X_i)[j]$ with $j' = X_S[preorder_{\mathcal{T}_S}(u)]$, and then find the internal node $x \in \mathcal{T}_\mathcal{G}$ with label $X_{j'}$. This is done with $x = node(select_{j'}(X, 1))$, which is computed in constant time as in Lemma 1. Once x is found, with children $x_1, x_2, \ldots$ in $\mathcal{T}_\mathcal{G}$, we recursively expand $x_2, \ldots$ by computing each label $X[\mathit{preorder}_\mathcal{G}(x_k)]$ and mapping it back to $\mathcal{T}_S$.

We stop the process when we extract the first ℓ symbols. Charging the cost to the new symbol to be expanded, and since there are no unary paths, it follows that we carry out $O(\ell)$ steps [GKPS05]. All costs per step are $O(1)$ except for the $O(1/\epsilon)$ to access X_S^{-1} and X (recall the end of the proof of Lemma 2). The time is therefore $O(\ell/\epsilon)$. $\square$

For extracting suffixes of rules in $\mathcal{G}$, we need another version of $\mathcal{T}_S$ that stores the rightmost paths. The spaces we have given then double.

Example. Returning to Figure 2, let us now extract $\mathcal{F}(X_2)[1..5]$. After extracting $\mathcal{F}(X_2)[1] = a$, we continue with $u = \mathit{level\text{-}ancestor}_{\mathcal{T}_S}(v, 2)$, which is labeled X_4. We find the internal node $x \in \mathcal{T}_\mathcal{G}$ with label X_4 (see Figure 1) and see that its second to last children in $\mathcal{T}_\mathcal{G}$ are labeled X_6, X_1, X_3. Those represent single symbols (since $C[6] = C[1] = C[3] = 1$), which we obtain with $\mathit{rank}_1(C, 6) = 4 = l$, $\mathit{rank}_1(C, 1) = 1 = a$, and $\mathit{rank}_1(C, 3) = 2 = b$. We have then obtained $\mathcal{F}(X_2)[1..4] = alab$, but still need $\mathcal{F}(X_2)[5]$. For this sake we continue with $u = \mathit{level\text{-}ancestor}_{\mathcal{T}_S}(v, 1)$, which is labeled X_9. We find the internal node $x \in \mathcal{T}_\mathcal{G}$ with label X_9 and see that its second and last child is labeled X_8. We then map X_8 back to $\mathcal{T}_S$ and find that its ancestor that is a child of the root is labeled X_1, which gives us again $\mathcal{F}(X_2)[5] = a$.

In practice. Our experiments will show that the optimal-time extraction does not make much difference with respect to a simple variant of the procedure of Section 4.1: To extract a prefix (suffix) of length ℓ of $\mathcal{F}(X_i)$, we recursively expand its children left-to-right (right-to-left) until obtaining ℓ symbols. This requires $O(\ell)$ time for the extracted symbols plus $O(h)$ time to explore the paths that are not fully expanded, for a total of $O(\ell + h)$ time.

4.3 Extracting Arbitrary Substrings

In order to extract any given substring of T, we add a bitmap $L[1..n]$ that marks with a 1 the first position of each T_i (i.e., expansion of a grammar tree leaf, recall Definition 2) in T; see the bottom of Figure 1. We can then compute in constant time the starting position $p(v)$ in T of any grammar tree node v: we compute the number k of the grammar tree leaves preceding v, $k = \mathit{leafrank}(v)$, and then $p(v)$ is the starting position of the next leaf:

$$p(v) = \mathit{select}_1(L, \mathit{leafrank}(v) + 1).$$

To extract $T[p..p+\ell-1]$, we binary search the children $u_1, u_2, \ldots$ of the root of $\mathcal{T}_\mathcal{G}$ for the child u_j covering position p, that is, $p(u_j) \le p < p(u_{j+1})$. If $u = u_j$ is an internal node, we continue recursively with its children. Instead, if u is a leaf representing a nonterminal X_i, we go to the internal node $v \in \mathcal{T}_\mathcal{G}$ with label X_i as we did in Lemma 1, translate position p to the area below the new node v (i.e., p becomes $p - p(u) + p(v)$), and continue recursively from v. At some point we reach a leaf u, representing a terminal $X_i \rightarrow a$, that covers the position p (indeed, $a = T[p]$). We repeat an analogous process for the final position, $p' = p + \ell - 1$, reaching the leaf u'.

We have then obtained the paths of the parse tree that lead to the leaves u and u', which represent the text positions p and p'. We now extract $T[p..p+\ell-1]$ with a standard technique, see for example Bille et al. [BLR+15]. Let u^* be the lowest common node in the paths from the root to u and u' (i.e., where the paths diverge). For every parent-child edge (v, v') in the path from u^* to u, we collect all the children of v to the right of v'. Similarly, we collect all the children of v to the left of v' in the path from u^* to u'. We also collect u, u', and the children of u^* between those leading to u and u'. All the subtrees of those nodes are expanded as in Section 4.1, in increasing DFS order of the parse tree.

Lemma 5. *The time of the described extraction procedure is $O(\ell + h \lg(G/h))$.*

Proof. The total time is $O(\ell)$, which is the sum of the sizes of the expanded trees, plus the time to find u and u'. These are at depth at most h and we perform a binary search each time we descend, so the total time is $O(h \lg G)$. Note, however, that we cannot repeat nodes in a path because the grammar has no cycles, so we binary search at most h distinct right-hand sides that add up at most to length G. The worst case, by Jensen's inequality, is that each searched right-hand side is of length G/h, leading to a maximum time complexity of $O(\ell + h \lg(G/h))$. $\square$

The number of 1s in L is at most G. Since we only need $select_1$ on L, we can use the compressed bitmap representation of Section 2.1, which supports the operation in constant time and requires $G \lg(n/G) + O(G)$ bits.

In practice. We also implement an alternative technique, which is less attractive in the worst case but is faster in practice. Instead of doing successive binary searches in the path along internal nodes u until finding a grammar tree leaf (and then switching to its definition v), we directly compute the grammar tree leaf with $rank_1(L, p)$. We can still, in the worst case, switch h times, and since computing $rank$ on L takes time $O(\lg \min(G, n/G))$, the worst-case time is $O(\ell + h \lg \min(G, n/G))$.

More precisely, we first compute $r_1 = rank_1(L, p)$ and $r_2 = rank_1(L, p + \ell - 1)$, so that the area to extract intersects from the r_1th to the r_2th leaves of $\mathcal{T}_G$. If $r_1 < r_2$, then we extract the suffix of length $select_1(L, r_1 + 1) - p$ of $X'[r_1]$, then extract the whole nonterminals $X'[r_1 + 1], \ldots, X'[r_2 - 1]$, and finally extract the prefix of length $p + \ell - select_1(L, r_2)$ from $X'[r_2]$.

If $r_1 = r_2$, instead, the substring is inside a single leaf $u \in \mathcal{T}_G$, whose label is X_i, $i = X'[r_1]$. We then find the internal node $v \in \mathcal{T}_\mathcal{G}$ labeled X_i as in Lemma 1 and continue by extracting $T[p - p(u) + p(v)..p - p(u) + p(v) + \ell - 1]$.

5 Locating Patterns

We now consider the problem of locating all the occurrences in T of a given pattern $P[1..m]$. In Definition 2, we partitioned T into blocks $T = T_1 \cdots T_{G+1-g+\sigma}$ according to the leaves of $\mathcal{T}_\mathcal{G}$. We now use this partition to classify the possible occurrences of P in T into *primary* and *secondary*, following a seminal idea by Kärkkäinen [Kär99].

Definition 3. *We say that an occurrence of a pattern P is* primary *with respect to the partition $T_1 \cdots T_{G+1-g+\sigma}$ if it spans more than one T_j. The other occurrences, completely inside some T_j, are called* secondary*.*

The strategy [Kär99] is to first find the primary occurrences and then induce the secondary ones from those. To find the primary occurrences of $P = p_1 p_2 \dots p_m$, Kärkkäinen considers each of the $m-1$ cuts $P = P_1 \cdot P_2$, $P_1 = p_1 \cdots p_i$ and $P_2 = p_{i+1} \cdots p_m$, for $1 \leq i < m$, and finds the text blocks ending with P_1 that are followed by P_2. His blocks correspond to Lempel-Ziv phrases, whereas our text partition is induced by the grammar tree leaves. We now prove that our strategy is valid; we start with the following definition.

Definition 4. *The* locus *of a primary occurrence $T[p..p+\ell-1]$ that spans blocks $T_j \cdots T_{j'}$ is the lowest node of $\mathcal{T}_G$ whose descendants include the jth to the j'th leaves.*

It is clear that, if v^* is the locus of a primary occurrence of P, and it is labeled X_i, then P occurs in $\mathcal{F}(X_i)$. Further, P occurs in $\mathcal{F}(X_j)$ for all the labels X_j of the ancestors of v^* in the grammar tree. Those correspond to the same occurrence of P in T, but P also appears in $\mathcal{F}(X_j)$ for every other leaf of $\mathcal{T}_G$ labeled X_j, and recursively in the other leaf occurrences of its ancestors, and so on. The next lemma proves that those are *all* the secondary occurrences of P in T.

Lemma 6. *Every secondary occurrence of P in T appears in a leaf $u \in \mathcal{T}_G$ labeled X_i, where the (only) internal node v labeled X_i is an ancestor of the locus v^* of a primary occurrence, or it is an ancestor of another leaf containing a secondary occurrence.*

Proof. We proceed by induction on $|\mathcal{F}(X_i)| \geq |P|$. Since P occurs inside $\mathcal{F}(X_i)$, it also occurs in the concatenation of the expansions of the leaves descending from v. If that occurrence spans more than one leaf, then that is a primary occurrence of P and its locus v^* descends from v. Because the right-hand sides of our modified grammar are of length at least two, this is the only possibility if $|\mathcal{F}(X_i)| = |P|$, which proves the base case of the induction. Otherwise, the occurrence is inside a leaf u' descending from v and labeled X_j, with $|\mathcal{F}(X_j)| < |\mathcal{F}(X_i)|$, which then contains a secondary occurrence of P. By the inductive hypothesis, the only internal node v' labeled X_j is an ancestor of the locus of a primary occurrence or of another leaf containing a secondary occurrence. $\square$

Our strategy is then to find the locus of each primary occurrence and then propagate it to all the other secondary ones. To efficiently find the locus of $P_1 \cdot P_2$, we will look for the nonterminal $X_k \rightarrow X_{k_1} X_{k_2} \dots X_{k_r}$ such that P_1 is a suffix of some $\mathcal{F}(X_{k_i})$ and P_2 is a prefix of $\mathcal{F}(X_{k_{i+1}}) \cdots \mathcal{F}(X_{k_r})$. The secondary occurrences are then tracked in the grammar tree from X_k.

Although this scheme does not consider the case $m = 1$ (i.e., $P = p_1$), we handle it by finding its corresponding nonterminal $X_j \rightarrow p_1$ with $j = select_1(C, p_1)$, finding all the occurrences of X_j in $\mathcal{T}_G$ using $select_j(X, \cdot)$, and treating those leaves as the loci of primary occurrences.

5.1 Finding Primary Occurrences

We store a binary relation $\mathcal{R} \subseteq A \times B$ to find the primary occurrences. It has g rows labeled $X_1, \dots, X_g$, so $B = \mathcal{X}$, and $G - g + \sigma \leq G$ columns. Each column corresponds to a border between two consecutive text blocks, in a way that identifies the corresponding locus: there is one column per proper suffix $\alpha_i[j+1..r_i] = X_{i_{j+1}} \cdots X_{i_{r_i}}$ of a distinct rule $X_i \rightarrow \alpha_i = X_{i_1} \cdots X_{i_{r_i}}$. The labels belong to $[1..G+1]$. The relation contains one pair per column: $(\alpha_i[j], \alpha_i[j+1..r_i]) \in \mathcal{R}$ for all $1 \leq i \leq g$ and $1 \leq j < r_i$. Its label is the preorder of the $(j+1)$th child of the internal node that defines X_i in $\mathcal{T}_G$. The space for the binary relation is $(G - g + \sigma)(\lg g + \lg G) + o(G \lg g)$ bits; recall Section 2.2.

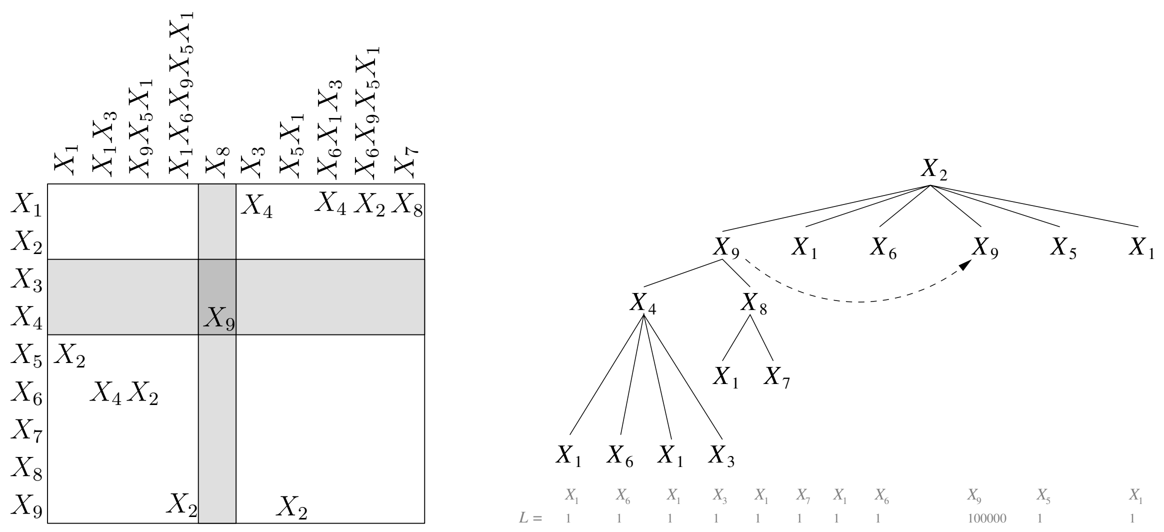


Fig. 3. Relation $\mathcal{R}$ for the grammar presented in Figure 1 (the grammar tree is replicated on the right). The highlighted ranges correspond to the result of searching for $b \cdot ar$, where the single primary occurrence corresponds to the locus node labeled X_9. The dashed arrow shows a secondary occurrence of X_9 that is found from the primary one.

Recall that, in our preprocessing of Section 3, we have sorted $\mathcal{X}$ according to the lexicographic order of $\mathcal{F}(X_i)^{rev}$. We also sort the suffixes $\alpha_i[j+1..r_i]$ lexicographically with respect to their expansion, that is, $\mathcal{F}(\alpha_i[j+1]) \cdot \mathcal{F}(\alpha_i[j+2]) \cdots \mathcal{F}(\alpha_i[r_i])$. This can be done as when sorting $\mathcal{X}$, in $O(G \lg G \lg n)$ time and $O(G \lg^2 n)$ bits of space: we produce a binary grammar of $O(G)$ rules so that $X_i \rightarrow X_{i_1} X_{i_2} \cdots X_{i_{r_i}}$ is replaced by $X_i \rightarrow X_{i_1} X_{i,2}$, $X_{i,2} \rightarrow X_{i_2} X_{i,3}$, and so on until $X_{i,r_i-1} = X_{i_{r_i-1}} X_{i_{r_i}}$; note that $\mathcal{F}(X_{i,j+1})$ is the expansion of $\alpha_i[j+1..r_i]$ if we also assume $X_{i,r_i} = X_{r_i}$. We then compute in time $O(G \lg(n/G))$ a compressed longest-common-extension data structure [I17] on the binary grammar and then use any sorting algorithm, as in Section 3. The lexicographic position of $\mathcal{F}(X_{i,j+1})$ then corresponds to that of the expansion of $\alpha_i[j+1..r_i]$.

Example. Figure 3 illustrates how $\mathcal{R}$ is defined and used for the grammar presented in Figure 1. For simplicity, each point in the binary relation shows the label of the locus node, although we actually store the preorder of the first child of the corresponding rule suffix.

Given P_1 and P_2, we first find the range of rows whose expansions finish with P_1, by binary searching for P_1^{rev} in the expansions $\mathcal{F}(X_i)^{rev}$; note that i is precisely the row number we access. Each comparison in the binary search needs to extract up to $|P_1|$ terminals from the suffix of $\mathcal{F}(X_i)$. As shown in Section 4.2, this can be done in $O(|P_1|/\epsilon)$ time. Similarly, we binary search for the range of columns whose expansions start with P_2. Each comparison needs to extract up to $\ell = |P_2|$ terminals from the prefix of $\mathcal{F}(\alpha_i[j+1]) \cdot \mathcal{F}(\alpha_i[j+2]) \cdots$. Let c be the column we wish to compare to P_2. We extract the label, that is, the preorder $p = S_{\mathcal{L}}[c]$ associated with the column, in constant time (Section 2.2), and compute the corresponding node, $v = node(p)$. We then extract the first ℓ symbols from v as done in Section 4.2. If v expands to less than ℓ symbols, we continue with *nextsibling*(p), and so on, until we extract ℓ symbols or we exhaust the suffix of the rule. This requires in total

$O(|P_2|/\epsilon)$ time. Thus our two binary searches require time $O((m/\epsilon)\lg G)$ (in practice, we extract only the symbols needed to decide each lexicographic comparison during the binary search).

This time can be further improved by building a trie of sampled expansions, P_A and P_B, for columns and rows of $\mathcal{R}$. We sample expanded strings at regular intervals and store them in a Patricia tree [Mor68]. We first search for the pattern in the Patricia tree, and then complete the process with a binary search between two sampled strings (we first verify the correctness of the Patricia search by checking that our pattern actually prefixes any string in the range found). By sampling one out of $\lg n$ strings, the search time becomes $O((m/\epsilon)\lg\lg n)$ and we only require $O(G)$ bits of extra space, since the Patricia trees need $O(\lg n)$ bits per node.[6]

Once we identify a range of rows $[a_1, a_2]$ and of columns $[b_1, b_2]$, we retrieve all the k points in the rectangle and their labels in time $O((k+1)(1+\lg g/\lg\lg G))$. The nodes $node(p)$, for each preorder p labeling a point in the range, are the children of the loci of primary occurrences, so that P_1 is the suffix of the expansion of the previous sibling of p and P_2 is the prefix of the expansion of p and its following siblings.

We have to carry out this search for $m-1$ partitions of P, whereas each primary occurrence is found exactly once. Adding the space of Theorem 2 and the space of $\mathcal{R}$, P_A and P_B (i.e., all the structures in Table 3), yields the following result.

Lemma 7. *The loci of the occ primary occurrences of P are found in time $O((m^2/\epsilon)\lg\lg n + (m + occ)(1+\lg g/\lg\lg G))$ with a structure that uses $G\lg n + 2G\lg g + \epsilon\, g\lg g + o(G\lg g) + O(G)$ bits of space.*

Example. Figure 3 shows how we find the only primary occurrence of $P_1 \cdot P_2 = b \cdot ar$ in our example grammar. Nonterminals X_3 and X_4, which expand to b and $alab$, respectively, are those ending with $P_1 = b$, whereas X_8 is the only suffix whose expansion is prefixed by $P_2 = ar$ (indeed, $\mathcal{F}(X_8) = ar$). The grid shows, conceptually, the locus node of that primary occurrence (the internal one labeled X_9), though it actually stores the preorder of its second child, labeled X_8.

Note that, if P occurs several time in the expansion of a locus node v^*, then the same v^* will be the parent of various node preorders p; it can even appear several times for the same p with different cuts $P_1 \cdot P_2$. For example, $P = aaa$ appears 4 times with locus $X_1 \rightarrow X_2X_2X_2$ and $X_2 \rightarrow X_aX_a$, twice with the partition $P_1 \cdot P_2 = a \cdot aa$ and twice with $P_1 \cdot P_2 = aa \cdot a$; two of those with p corresponding to the second X_2 and two with p corresponding to the third X_2. We show next how to report primary and secondary occurrences starting from those nodes $node(p)$.

5.2 Tracking Secondary Occurrences through the Grammar Tree

The remaining problem is how to track all the secondary occurrences triggered by a primary occurrence, and how to report the positions where they occur in T. Given a primary occurrence for partition $P = P_1 \cdot P_2$ located at the child $v = node(p)$ of the locus $v^* = parent(v)$, the starting position of P in T is $p(v) - |P_1|$; recall the formula to compute $p(\cdot)$ in the beginning of Section 4.3. We must, however, walk the upward path from v^* to the root to find the secondary occurrences. For every ancestor u in the path, we obtain its nonterminal label X_i, $i = X[preorder(u)]$, in time

[6] We could push it a bit further, for example sampling one out of $\lg n \lg\lg g/\lg g$ strings to obtain $o(G\lg g) + O(G)$ bits of extra space and a search time of $O\left((m/\epsilon)\lg\left(\frac{\lg n \lg\lg g}{\lg g}\right)\right)$, but we opt for a simpler formula.

$O(1/\epsilon)$ and look for all the other occurrences of i in X, $u' = \mathit{leafselect}(\mathit{select}_i(X', \cdot))$, each in time $O(\lg\lg g)$ (recall Section 4.1). For each such leaf u', we report the corresponding occurrence of P, $p(v) - |P_1| + p(u') - p(u)$, and recursively walk the path from u' to the root.

Lemma 8. *Once the loci of the primary occurrences are located, the occ occurrences of P in T are obtained, with the structures already defined, in $O(\mathit{occ}\,(1/\epsilon + \lg\lg g))$ time.*

Proof. It seems that we do $O(h)$ steps per occurrence reported, since we walk the upward path to the root for each. The steps, however, amortize to $O(1)$. Our preprocessing of Section 3 guarantees that every nonterminal X_i appears at least twice in the grammar tree. Therefore, every time we move from an ancestor u of v (or of u') to its parent, we know that u is an internal node and thus X_i $(i = X[u])$ appears at least once more in the grammar tree. The cost of walking through u in the path from v^* (or u') to the root can then be charged to the first leaf labeled X_i (the one that is found at $\mathit{select}_i(X', 1) = \mathit{select}_i(X, 2)$). The cost of each step is $O(1/\epsilon + \lg\lg g)$. $\square$

Example. In Figure 3, once we find the primary occurrence of $b \cdot ar$ at the internal node v labeled X_8, we compute $\mathit{leafrank}(v) = 4$ and obtain the starting position of v in T, $p(v) = \mathit{select}_1(L, 4 + 1) = 5$. Since $|P_1| = 1$, the position of this occurrence is $5 - 1 = 4$, that is, $T[4..6] = \mathit{bar}$. Further, starting from the locus node $u = v^*$, labeled X_9, we walk up to the root. We then find the only secondary occurrence because X_9 occurs again at a leaf u' of $\mathcal{T}_\mathcal{G}$. This occurrence is at position $p(v) - |P_1| + p(u') - p(u) = 5 - 1 + 9 - 1 = 12$, and thus $T[12..14] = \mathit{bar}$ is the second occurrence.

6 The Resulting Index

By adding up the spaces of Table 3, the query times of Lemmas 7 and 8, and the construction time and space of Sections 3 and 5.1, we have our central result, Theorem 1, where for simplicity we have replaced the cost per occurrence of $1/\epsilon + \lg\lg g + \lg g/\lg\lg G$ by just $1/\epsilon + \lg g/\lg\lg g$.

By using $\epsilon = \Theta(1)$ and $\epsilon = 1/\lg\lg g$, we obtain two simpler results. From the first we can adjust ϵ to obtain upper bounds of $G\lg n + (2+\epsilon)G\lg g$ bits in space and $O((m^2 + \mathit{occ})\lg G)$ in time because $\lg\lg n \le \lg G$; this is the simplified result we give in the abstract.

Corollary 2. *Let a sequence $T[1..n]$ be represented by a context-free grammar with g symbols and size G. Then, for any constant $0 < \epsilon \le 1$, there exists a data structure using at most $G\lg n + 2G\lg g + \epsilon\, g\lg g + o(G\lg g) + O(G)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O(m^2\lg\lg n + (m + \mathit{occ})\lg g/\lg\lg g)$.*

Corollary 3. *Let a sequence $T[1..n]$ be represented by a context-free grammar with g symbols and size G. Then, there exists a data structure using at most $G\lg n + 2G\lg g + o(G\lg g) + O(G)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O(m^2\lg\lg n\lg\lg g + (m + \mathit{occ})\lg g/\lg\lg g)$.*

Finally, by using a geometric structure [CLP11] that uses $O(G\lg G)$ bits for the binary relation, we can report the k points in a range in time $O((k+1)\lg^\epsilon g)$ for any constant $\epsilon > 0$. With $O(G\lg n)$ bits we can also use a dense sampling for our Patricia trees P_A and P_B, which allow searching for all the pattern cuts in time $O(m^2)$. We then obtain a faster structure, Corollary 1.

7 Implementation and Experiments

7.1 Implementation

We implemented our grammar-based self-index on top of SDSL (*Succinct Data Structures Library*)[7], which is written in C++11 and contains efficient deployments of several succinct data structures. We implement our structures as follows:

C, L: In repetitive collections it holds that $g \leq G \ll n$; we also expect that $\sigma \ll g$ for large texts. It follows that bitmaps C (of length g and with σ 1s) and L (of length n and with less than G 1s) are expected to be sparse. We then represent them using the class `sd_vector` from SDSL, which implements Sadakane and Okanohara's sparse array [OS07].

π, X_S: For the permutations π and X_S (which is represented plainly, not with X'_S and B_S), we use the class `inv_permutation_support<`t`>` of SDSL, which gives access to the inverse permutation in at most t steps, and fix t to 32.

X' : The sequence X' is represented using the structure of Golynski et al. [GMR06] (`wt_gmr` in SDSL), which is advisable in our case, where the alphabet of X' is large and the sequence is almost incompressible. This is an implementation of the theoretical version [BN15] we use in Section 2.1, which has constant-time *select*, $O(\lg g)$ time for *rank*, and $O(t)$ time for *access*, with an inverse permutation with parameter t (we use the same $t = 32$ here).

$\mathcal{R}$: Our representation for $\mathcal{R}$ is the same structure used in the implementation of Claude and Navarro [CN10] for labeled binary relations, based on wavelet trees [Nav14]. The sequence $S_{\mathcal{L}}$ is represented in plain form.

$\mathcal{T}_{\mathcal{G}}$, P_A, P_B, $\mathcal{T}_S$: The topology of the grammar tree $\mathcal{T}_{\mathcal{G}}$ and of the sampled Patricia tries P_A and P_B is represented with a variant of balanced FF (Section 2.3) called DFUDS [BDM$^+$05], which is faster in practice for moving towards children. The trees $\mathcal{T}_S$, instead, are represented using FF [NS14], which is more efficient for level ancestor queries. Both are implemented over the parentheses support of SDSL (`bp_support_sada`).

To generate the grammar $\mathcal{G}$ we use the RePair algorithm [LM00], in particular Navarro's implementation[8]. RePair produces a binary grammar (i.e., all the rules have 2 symbols in their right-hand side) plus a long initial rule. We then postprocess the resulting grammar as required for our index, see Section 3.

We test four versions of our index, called **g-index** in the experiments:

- The variants whose name continue with **binary_search** use plain binary search on the rules prefixes/suffixes in order to find the row and column intervals on the grid.
- The variants whose name instead continue with **patricia_tree** speed up this process using the sampled Patricia trees P_A and P_B, which take one string every 4, 8, 16, 32, and 64 positions.
- The variants suffixed **trie** use the algorithm of Gasieniec et al. [GKPS05] (Section 4.2) to extract rule prefixes/suffixes in optimal time, using the two variants of $\mathcal{T}_S$.
- The variants suffixed **notrie**, instead, omit the structures $\mathcal{T}_S$ and extract the text from the rules in recursive form.

[7] `https://github.com/simongog/sdsl-lite`
[8] `http://www.dcc.uchile.cl/gnavarro/software/repair.tgz`

– The term **qgram** indicates that we add a short q-gram ($q = 2, 4, 6, 8, 10, 12$) with the prefix and suffix of the expansion of each nonterminal [CFMPN16], to speed up extraction during binary searches. The strings are stored in a dictionary compressed with Huffman and Front Coding. Since the q-grams are limited, the binary search must be completed, either using plain decompression of nonterminals (suffix **dfs**), the real-time prefix extraction (suffix **trie**), or plain decompression speeded up by using the same q-grams for the first symbols of each nonterminal we must decompress (suffix **smp**).

Our implementation is available at `https://github.com/apachecom/grammar_improved_index/`.

7.2 Experimental Setup

The experimental evaluation was carried out using the environment provided in *Pizza&Chili* (`http://pizzachili.dcc.uchile.cl`). We compared our implementation with the available indexes in the state of the art that are most faithful with respect to different compressibility measures:

slp-index[9] is the only previous implementation of a grammar-based index [CN10], using $O(G \lg n)$ bits like ours. It does not guarantee, however, logarithmic locating time per occurrence. It uses the same RePair algorithm we use to build the index (a construction over the heuristically balanced version of RePair is called **slp-index-bal**). In its optimized version [CFMPN16], it speeds up the binary searches by storing the q-gram prefixes of the strings expanded by each nonterminal, as we use in the qgram variant of g-index, yet here the best values are $q = 4, 8, 16$.

lz-index[10] is the only implementation of a Lempel-Ziv based index [KN13] that guarantees $O(z \lg n)$ bits of space on a Lempel-Ziv parse of z phrases. We also include the LZ-End variant, **lz-end-index**. We use their optimized implementations [CFMPN16], which were shown to outperform slp-index both in space and time.

r-index[11] is the only implementation of a classical self-index (i.e., suffix-array based) using $O(r \lg n)$ bits, where r is the number of runs in the Burrows-Wheeler Transform of the text [GNP20].

We use six real repetitive collections from Pizza&Chili repetitive corpus[12]. Three of these collections contain DNA sequences extracted from different sources: *para* and *cere* are extracted from the Saccharomyces Genome Resequencing Project[13], whereas *influenza* is formed by DNA sequences of H. Influenzae taken from the National Center for Biotechnology Information (NCBI)[14]. Collection *einstein.en* is formed by all version of the articles in English of Albert Einstein taken from Wikipedia. Collections *kernel* and *coreutils* are formed by all versions 1.0.x and 1.1.x of the Linux Kernel[15], and all versions 5.x of the Coreutils package[16], respectively. Table 4 lists their main characteristics and how they compress under different measures; note how our grammar preprocessing reduces G and g significantly, making G 1.7–2.5 times larger than z and 1.4–2.9 times smaller than r.

[9] `https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/SLP`
[10] `https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/LZ`
[11] `https://github.com/nicolaprezza/r-index`
[12] `http://pizzachili.dcc.uchile.cl/repcorpus/real`
[13] `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp`
[14] `http://www.ncbi.nlm.nih.gov`
[15] `https://mirrors.edge.kernel.org/pub/linux/kernel`
[16] `https://ftp.gnu.org/gnu/coreutils`

Table 4. Main characteristics of the collections: n (size in bytes), σ (alphabet size), z (number of Lempel-Ziv phrases), r (number of runs in the BWT), G/g – repair (size of the RePair grammar and its number of symbols), and G/g – proc (the same after applying the transformations of Section 3).

Collection	n	σ	z	r	G – repair	g – repair	G – proc	g – proc
para	429,265,758	5	2,332,908	15,636,740	7,338,520	3,093,540	5,344,480	1,099,500
cere	461,286,644	5	1,700,859	11,574,641	5,780,080	2,561,120	4,069,450	850,491
influenza	154,808,555	15	770,253	3,022,822	2,174,650	642,965	1,957,370	425,697
einstein.en	467,626,544	139	91,036	290,239	263,962	100,763	212,903	49,843
kernel	257,961,616	162	794,290	2,791,368	2,185,860	1,058,260	1,374,650	247,212
coreutils	205,281,778	235	1,446,891	4,684,465	3,798,100	1,822,380	2,409,460	433,854

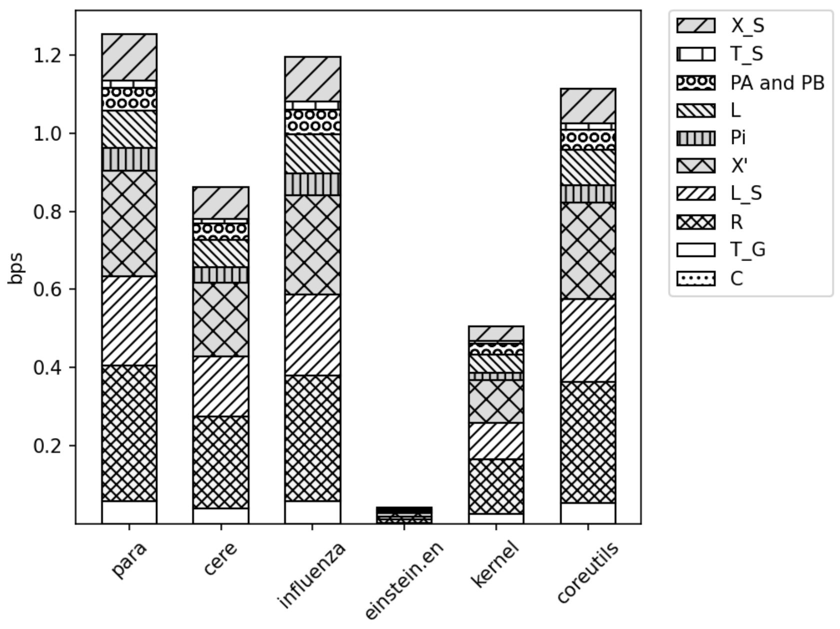


Fig. 4. Space breakdown of our index, in bits per symbol (bps), for all the text collections.

Our times per occurrence are the average over 1000 patterns of length 10 extracted at random from each collection. Our extraction times average over 1000 queries at random text positions.

7.3 Space Breakdown

Figure 4 shows the space used by each component of g-index, assuming a sampling step of 8 for P_A and P_B. It can be seen that the bulk of the space is contributed, in about equal parts, by X', $\mathcal{R}$, and $\mathcal{L}_S$. The components L and X_S come next in terms of impact, and π, the tree topologies $\mathcal{T}_G$ and $\mathcal{T}_S$, and the Patricia trees contribute the least, together with C (which cannot be seen in the plot).

This anticipates that it may be a good idea to pay for the extra space of P_A and P_B if they yield a good speedup, whereas it may be a good idea to get rid of $\mathcal{T}_S + X_S$ if they do not provide significant speedups, since this would free a noticeable amount of space.

7.4 Locating Time

We first discuss the results on our index and then compare it with the others.

Tuning our index. Figure 5 shows the space-time tradeoffs obtained for locating patterns of length 10 over all the g-index variants and parameter values on all the collections.

In all cases, the use of Patricia trees with a sufficiently sparse sampling rate can reach essentially the same space of the plain binary-search versions. Even with the sparsest sampling rate (1 out of 64), the Patricia trees outperform the binary searches, most sharply on the DNA alphabets. The rule samplings of the qgram versions also outperform binary searches on DNA alphabets without increasing the space, reaching the sweet point at value 8. Nevertheless, the Patricia trees always make better use of the space.

On the other hand, the use of the tries $\mathcal{T}_S$ increases the space by about 15% while providing only a slight improvement in time (the most noticeable improvements occur on *para* and *cere*). Still, it is always more convenient to use Patricia trees and no tries $\mathcal{T}_S$.

As a result, we take g-index-patricia_tree-notrie as the most convenient version of our implementation, and we call it simply g-index henceforth.

Comparing with other indexes. Figure 6 compares our chosen variant of g-index with the other index variants and parameter values. The use of denser samplings yields an interesting space-time tradeoff for g-index on DNA. On the other texts, it is better to use it with the sparsest sampling.

In all cases, with the sparsest sampling g-index uses almost the same space of the previous grammar-based index, slp-index (except on *influenza*, where g-index is 20% larger), and it is 1.75 to 6 times faster. This makes g-index the best grammar-based self-index in practice. Note that there are almost no differences between slp-index and slp-index-bal, which confirms that typical grammar heights do not affect extraction time in practice.

Index lz-index outperforms slp-index in both space and time, as in previous work [CFMPN16]. While losing to lz-index in space is expected because $z \leq G$ always holds, grammars allow for better methods to access the text. The index slp-index was, however, unable to take advantage of those methods to outperform lz-index in time. Now our g-index does offer a space-time tradeoff, typically using more space than lz-index, but in exchange being faster: for example, with sampling value 8, g-index is 50%–70% larger than lz-index, but around 30% faster (except only 15% faster on *cere*). Instead, with the sparsest sampling, g-index matches lz-index almost exactly on *kernel* and outperforms it both in space and time on *coreutils*. The variant lz-end-index takes little more space than lz-index and is usually faster; it outperforms g-index on *cere*.

Finally, r-index is much faster than all the others, but also much larger (1.8–4.3 times larger than lz-index and 1.5–2.5 times larger than g-index).

Overall, g-index provides a relevant space/time tradeoff between the previous implemented indexes.

Varying the pattern length. Figure 7 shows how the locate time evolves with the pattern length m on *einstein.en* (the numbers in brackets are the sample values of the indexes), and consider pattern lengths 5, 10, 20, 30, 40, and 50.

The left plot shows that, while g-index is significantly faster than slp-index and lz-index on this text for small m, it slows down as m increases (the same happens for all the g-index variants, with

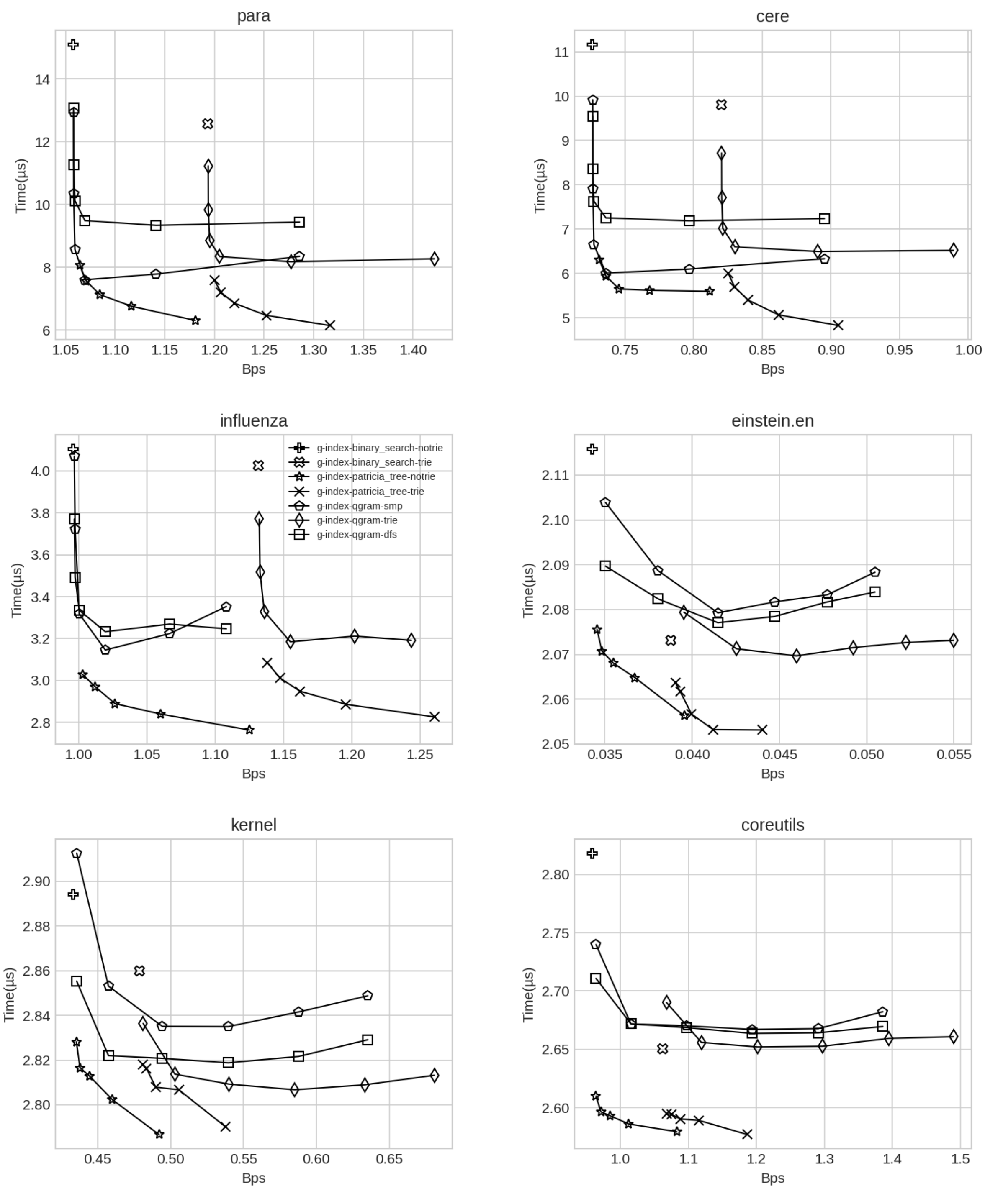


Fig. 5. Time-space tradeoffs for locating on different collections and variants of our index. The time is given in microseconds per occurrence and the space in bits per symbol (bps).

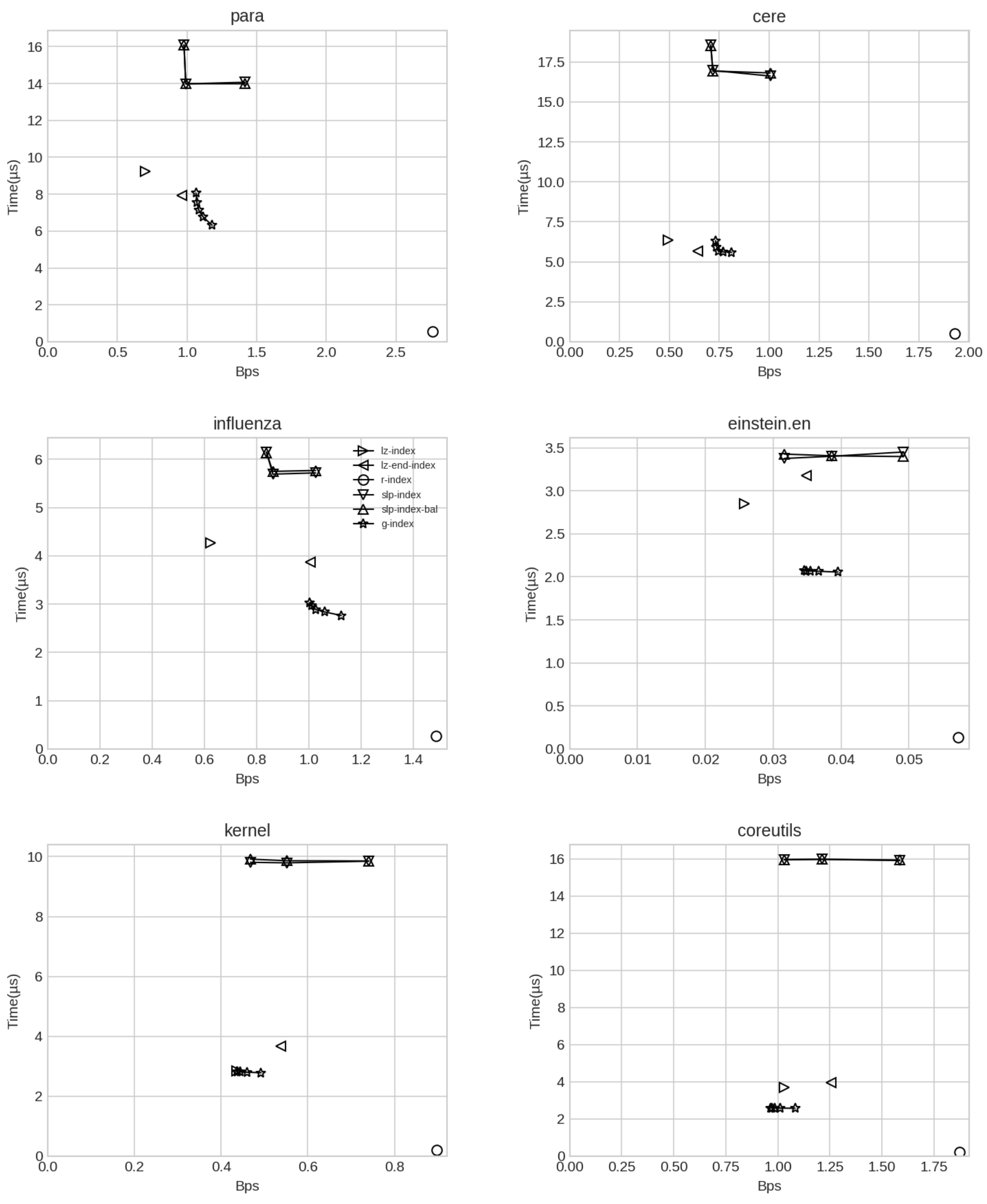
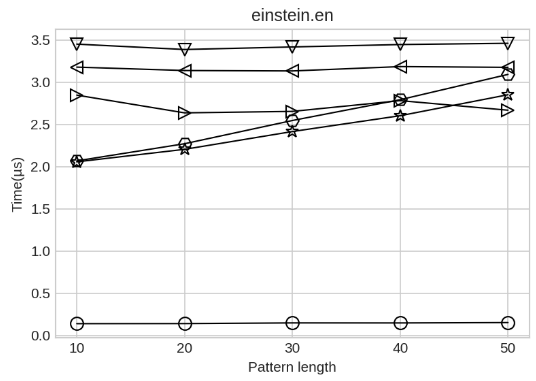


Fig. 6. Time-space tradeoffs for locating on different collections and indexes. The time is given in microseconds per occurrence and the space in bits per symbol (bps).



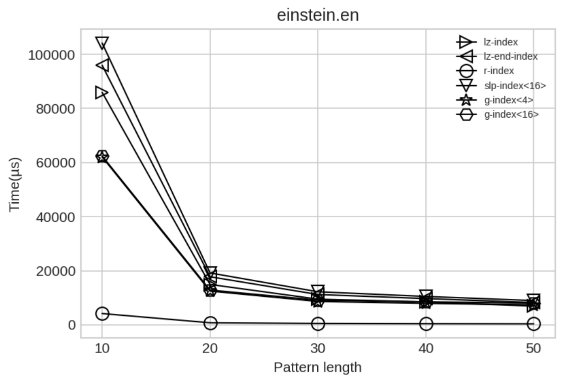


Fig. 7. Locating time for increasing pattern lenghts on *einstein.en*. The time is given in microseconds per occurrence on the left, and per pattern symbol on the right.

the notrie variants slowing down more clearly). Eventually, g-index loses to lz-index for $m = 50$. The most important lengths, however, where a large number of occurrences are found, are the short ones. The *total* query times are much less significant for long patterns, as shown on the right plot.

7.5 Extraction Time

Figure 8 shows the time per extracted symbol of the different indexes and collections, when extracting 100 consecutive text symbols. The r-index is excluded because it does not support this operation. Note that the extraction in our g-index is independent of whether or not we use binary search or Patricia trees. The variant that continues with **binary_search_leaf** descends from the root symbol, binary searching the children, to reach the desired substring to extract, as in the main body of Section 4.3. Instead, the variant that continues with **rank_phrases** uses *rank* on the bitmap L, so as to find faster the phrases to be expanded, as described at the end of Section 4.3. The suffixes **trie**/**notrie** refer again to using/not using the structure for extracting prefixes and suffixes in linear time (for the phrases that are not completely contained in the area to extract). Finally, if **qgram** follows g-index, we use the q-grams to speed up extraction, with lengths $2, 4, 6, 8$.

In this section we vary the parameter t used for all inverse permutations, which makes little difference in the locating experiments but has a significant impact on extraction, because we perform many accesses to X. All the variants of g-index are shown for values $t = 2$, 4, 8, 16, 32. The exception is the gram variant, which uses fixed $t = 32$.

As it can be seen, in all cases the rank_phrases variant is faster than binary_search_leaf (by a very slight margin in the case of notrie). Further, the notrie versions use less space and are faster than the trie variants. As a result, the best g-index variant, both in space and time, is g-index-rank_phrases-notrie. It is also apparent that lz-index excells in extraction, dominating every other alternative almost everywhere (except on *einstein.en*, the most repetitive collection, where the $\bar{h}$ value of the Lempel-Ziv parsing is very high; lz-end-index shows its robustness in this aspect). On the other hand, our best g-index variant competes with slp-index, dominating it on *kernel* and *coreutils*, being dominated one *influenza* and *einstein.en*, and with mixed results on *para* and *cere*.

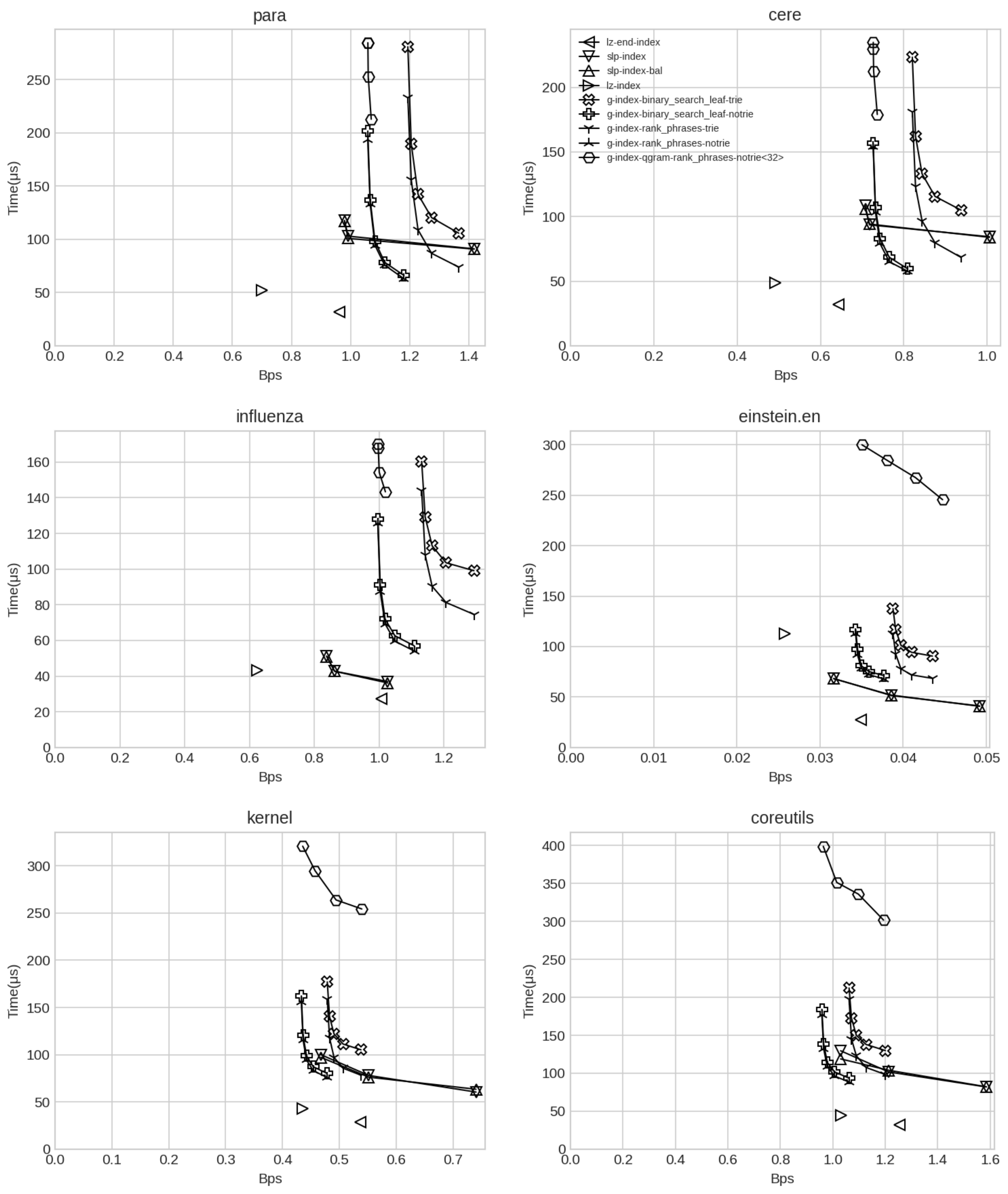


Fig. 8. Time-space tradeoffs for extracting on different collections and indexes. The time is given in microseconds per extracted symbol and the space in bits per symbol (bps).

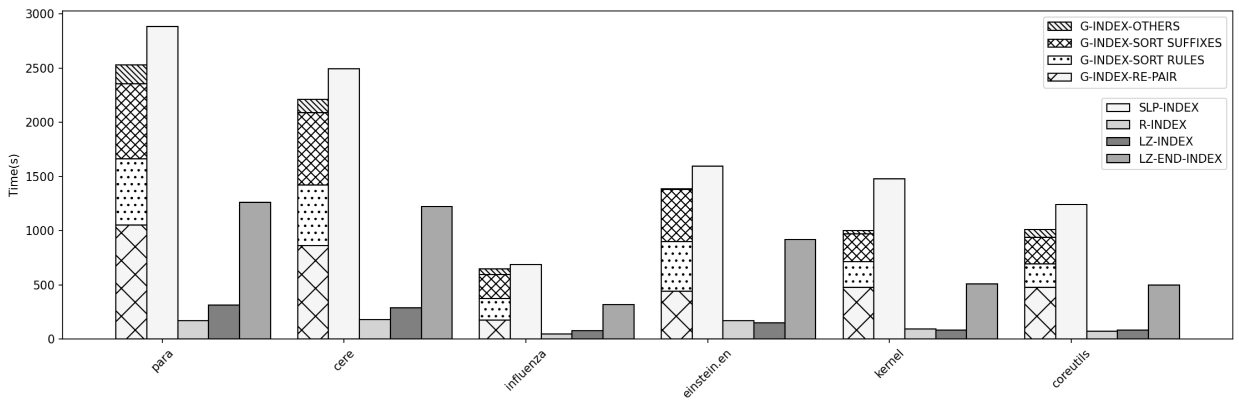


Fig. 9. Construction time breakdown of our index, in seconds (s), for all the text collections.

7.6 Construction

In this section we study the construction time and space of our index and compare them with those of the other indexes.

Figure 9 shows the construction time of our index, distinguishing the main parts: Running RePair to build the grammar (RE-PAIR), sorting the nonterminals $\mathcal{X}$ according to their expansions during our grammar preprocessing of Section 3 (SORT RULES), and sorting the coordinates of $\mathcal{R}$ according to the reverse rule and rule suffix expansions in Section 5.1 (SORT SUFFIXES).

Overall, our construction takes 3–6 seconds per megabyte of text. The sorting of rule expansions is done by reusing previous work [CN10]. This first uses $O(n)$ time and space to build the suffix and longest common prefix arrays of the text and a range minimum query data structure on the second array. With those, the rule expansions are then sorted in $O(G \lg G)$ time.

The figure also shows the construction time of the other indexes. The other grammar-based index, slp-index, is slower to build than g-index. The other indexes build considerably faster, with lz-end-index taking about half the time and r-index and lz-index being an order of magnitude faster.

Figure 10 shows a breakdown of the construction space. The figure includes the same stages as before, but we also distinguish the constructions of the structures that precede the sorting (PRE-SORT) from the sorting itself (SORT). The construction of our index requires significant space, around 15 times the text size, dominated by RE-PAIR and the structures of the PRE-SORT steps.

The figure also shows the construction space of the other indexes. The construction of slp-index is similar to ours and uses slightly more space. Instead, r-index uses much less space, and lz-index and lz-end-index use even less.

The construction space is certainly a bottleneck to use our index on larger text collections. A more sophisticated construction based on I's data structures [I17] would significantly reduce the space of sorting the rule expansions, to $O(G \lg^2 n)$ bits, with similar time complexity, $O(G \lg(n/G))$, as discussed in Sections 3 and 5.1. The remaining space bottleneck would be RE-PAIR, for which good approximations using little space have been recently developed [GIM+19] (we can also use other grammar construction algorithms, some of which are designed to handle very large texts too [TIS17], but RePair offers the best space in practice).

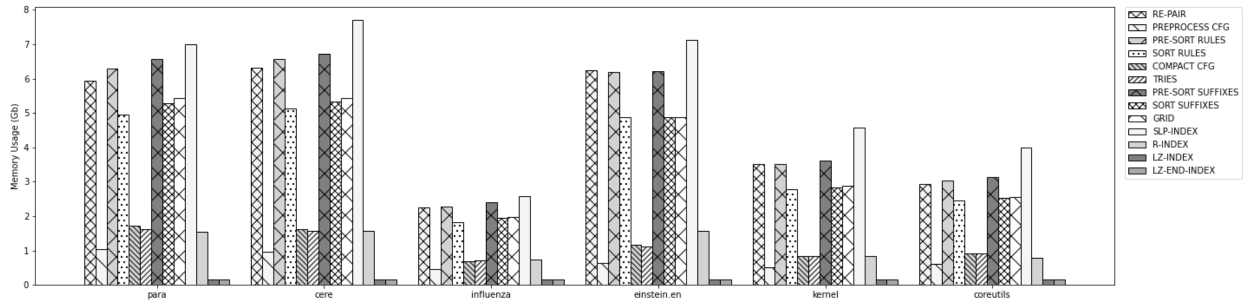


Fig. 10. Construction space of the successive steps of our index and of the other indexes, in gigabytes, for all the text collections.

8 Conclusions

We have presented the first compressed text index of space bounded by the size of an arbitrary context-free grammar and whose time per retrieved occurrence is logarithmic, independent of the grammar height. Given a text $T[1..n]$ represented by a grammar of size G, our index uses $G \lg n + (2+\epsilon) G \lg g$ bits of space, for any constant $\epsilon > 0$, and returns the *occ* occurrences of a pattern $P[1..m]$ in time $O((m^2 + occ) \lg G)$. We implemented our index and compared it with various alternatives in the literature, showing that it is practical and offers relevant space/time tradeoffs.

The most interesting open theoretical question is whether it is possible to obtain $O(G \lg G)$ bits, as obtained by grammar-based compressors, instead of $O(G \lg n)$, since in some text families it holds that $G = O(\lg n)$. This term owes to storing the lengths of the expansions of the nonterminals in bitmap L. We tried storing these lengths in the nonterminals, instead, and sampling the nonterminals that would store lengths. We then need to find a suitable sampling on the grammar DAG, which is the grammar tree where nonterminal leaves are identified with the internal node that defines them. Finding a suitable sampling on a DAG, however, is related to finding sparsest cuts in graphs [AHK04], which is not an easy problem.

With respect to practical results, an interesting research direction is to obtain competitive implementations of recent techniques that reduce the $O(m^2)$ term in the search complexity [GGK$^+$14,BEGV18,NP19,CE18,KNP20,CEK$^+$20]. Some of those methods [GGK$^+$14,BEGV18] have a penalty factor of $O(\lg \lg z)$ or $O(\lg(n/z))$ in the space, which is far from negligible, and thus they are unlikely to be space-competitive. Other indexes [CE18,CEK$^+$20] build on grammars and look more promising. They manage to ensure that P needs be cut into $P_1 \cdot P_2$ at only $O(\lg m)$ places in order to spot all the primary occurrences, which reduces the $O(m^2)$ term to $O(m)$. For this to hold, however, the grammar must be of a special type called locally-consistent. In our experience, RePair outperforms in space, by a wide margin, all the other grammar construction algorithms, including those that offer guarantees of the form $G = O(z \lg(n/z))$. Yet another class of indexes [NP19,KNP20] builds on block trees [BCG$^+$20] instead of grammars, reducing the $O(m^2)$ term to $O(m \lg n)$. Block trees have been shown to offer faster access than grammars in practice, though being competitive in space only when repetitiveness is very high [BCG$^+$20]. Finally, Christiansen et al. [CEK$^+$20, App. A] show an index of $O(G \lg n)$ bits that can be built on any grammar of size G, and reduces our $O(m^2)$ term to $O(m \lg n)$. Their space involves a constant that is likely to be significantly larger than ours, because they rely on a space-demanding data structure (the z-fast trie [BBPV18, Sec. H.2]) to search for t prefixes/suffixes of P in time $O(t \lg m)$ (indeed, these z-fast tries are indeed used in most of the recent work obtaining $o(m^2)$ search time). Therefore, various of these indexes could be faster, yet likely larger, than our grammar-based index, thereby providing new space/time tradeoffs between our index and the r-index [GNP20].

Acknowledgements

We thank the reviewers for their comments, which helped improve the presentation considerably; one of them pointed us the reference for efficient construction [I17].

References

[AB92] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 2nd Data Compression Conference (DCC)*, pages 279–288, 1992.

[AHK04] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\lg n})$ approximation to SPARSEST CUT $O(n^2)$ in time. In *Proc. 45th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 238–247, 2004.

[ANS12] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1):54–101, 2012.

[BBPV18] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. *CoRR*, abs/1804.04720, 2018.

[BCG$^+$15] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 26–39, 2015.

[BCG$^+$20] D. Belazzougui, M. Cáceres, T. Gagie, P. Gawrychowski, J. Kärkkäinen, G. Navarro, A. Ordóñez, S. J. Puglisi, and Y. Tabei. Block trees. *Journal of Computer and System Sciences*, 2020. To appear.

[BCN13] J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19–37, 2013.

[BCPT15] D. Belazzougui, P. H. Cording, S. J. Puglisi, and Y. Tabei. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, pages 142–154, 2015.

[BDM$^+$05] D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BEGV18] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theoretical Computer Science*, 713:66–77, 2018.

[BGG$^+$14] D. Belazzougui, T. Gagie, S. Gog, G. Manzini, and J. Sirén. Relative FM-indexes. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 52–64, 2014.

[BHH$^+$19] H. Bannai, M. Hirayama, D. Hucke, S. Inenaga, A. Jez, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. *CoRR*, 1908.06428, 2019.

[BLR$^+$15] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[BN15] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, 1994.

[CE18] A. R. Christiansen and M. B. Ettienne. Compressed indexing with signature grammars. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 331–345, 2018.

[CEK$^+$20] A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 2020. To appear.

[CFMPN10] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*, 2010.

[CFMPN16] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[CLL$^+$05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CLP11] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[CN10] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[CN12] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 180–192, 2012.

[CRA76] C. Cook, A. Rosenfeld, and A. Aronson. Grammatical inference by hill climbing. *Information Science*, 10:59—80, 1976.

[DJSS14] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[Eli74] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.

[Fan71] R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

[FM05] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[GGK$^+$12] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, pages 240–251, 2012.

[GGK$^+$14] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Theoretical Informatics Symposium (LATIN)*, pages 731–742, 2014.

[GIM$^+$19] T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, and Y. Takabatake. Rpair: Scaling up repair with rsync. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 35–44, 2019.

[GJL19] M. Ganardi, A. Jeż, and M. Lohrey. Balancing straight-line programs. In *Proc. 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1169–1183, 2019.

[GKPS05] L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th Data Compression Conference (DCC)*, page 458, 2005.

[GMR06] A. Golynski, J. I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GNP20] T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

[HLR16] D. Hucke, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 35–49, 2016.

[I17] T. I. Longest common extensions with recompression. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 18:1–18:15, 2017.

[Jez15] A. Jez. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[Jez16] A. Jez. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[Kär99] J. Kärkkäinen. *Repetition-Based Text Indexing.* PhD thesis, U. Helsinki, Finland, 1999.

[KMS$^+$03] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[KN13] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[KNP20] T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. In *Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN)*, 2020. To appear.

[KY00] J. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[MNSV10] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[Mor68] D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MRRR12] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[Nav14] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[Nav20] G. Navarro. Indexing highly repetitive string collections. *CoRR*, 2004.02781, 2020. To appear in *ACM Computing Surveys*.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NMWM94] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *Proc. 4th Data Compression Conference (DCC)*, pages 244–253, 1994.

[NP19] G. Navarro and N. Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

[NPC$^+$13] J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment: An efficient index for similar data. In *Proc. 24th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 337–348, 2013.

[NPL$^+$13] J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment: A practical index for similar data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 243–254, 2013.

[NS14] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[RO08] L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 11(4):359–388, 2008.

[Ryt03] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[SS82] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[Sto77] J. A. Storer. NP-completeness results concerning data compression. Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

[TIS17] Y. Takabatake, T. I, and H. Sakamoto. A space-optimal grammar compression. In *Proc. 25th Annual European Symposium on Algorithms (ESA)*, pages 67:1—67:15, 2017.

[VY13] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.