Improved Grammar-Based Compressed Indexes

Francisco Claude[1,⋆] and Gonzalo Navarro[2,⋆⋆]

[1] David R. Cheriton School of Computer Science, University of Waterloo.
[2] Department of Computer Science, University of Chile.

Abstract. We introduce the first grammar-compressed representation of a sequence that supports searches in time that depends only logarithmically on the size of the grammar. Given a text $T[1..u]$ that is represented by a (context-free) grammar of n (terminal and nonterminal) symbols and size N (measured as the sum of the lengths of the right hands of the rules), a basic grammar-based representation of T takes $N \lg n$ bits of space. Our representation requires $2N \lg n + N \lg u + \epsilon\, n \lg n + o(N \lg n)$ bits of space, for any $0 < \epsilon \leq 1$. It can find the positions of the *occ* occurrences of a pattern of length m in T in $O\left((m^2/\epsilon) \lg \left(\frac{\lg u}{\lg n}\right) + (m + occ) \lg n\right)$ time, and extract any substring of length ℓ of T in time $O(\ell + h \lg(N/h))$, where h is the height of the grammar tree.

1 Introduction and Related Work

Grammar-based compression is an active area of research that dates from at least the seventies. A given sequence $T[1..u]$ over alphabet $[1..\sigma]$ is replaced by a hopefully small (context-free) grammar $\mathcal{G}$ that generates just the string T. Let n be the number of grammar symbols, counting terminals and nonterminals. Let $N = |\mathcal{G}|$ be the *size* of the grammar, measured as the sum of the lengths of the right-hand sides of the rules. Then the grammar-compressed representation of T requires $N \lg n$ bits, versus the $u \lg \sigma$ bits required by a plain representation.

Grammar-based methods can achieve universal compression [20]. Unlike statistical methods, that exploit frequencies to achieve compression, grammar-based methods exploit repetitions in the text, and thus they are especially suitable for compressing highly repetitive sequence collections. These collections, containing long identical substrings, possibly far away from each other, arise when managing software repositories, versioned documents, transaction logs, periodic publications, and computational biology sequence databases. Good experimental results have been obtained by using grammar-based indexes [8].

Finding the smallest grammar $\mathcal{G}^*$ that represents a given text T is NP-complete [32, 6]. Moreover, the smallest grammar is never smaller than an LZ77 parse [34] of T. A simple method to achieve an $O(\lg u)$-approximation to the smallest grammar size is to parse T using LZ77 and then to convert it into a grammar [32]. A more sophisticated approximation [7] achieves ratio $O(\lg(u/|\mathcal{G}^*|))$.

⋆ Funded by Google U.S./Canada PhD Fellowship.
⋆⋆ Funded by Fondecyt Grant 1-110066, Chile.

While grammar-compression methods are strictly inferior to LZ77, and popular grammar compressors such as LZ78 [35], Re-Pair [23] and Sequitur [29], may yield sizes much larger than $|\mathcal{G}^*|$ [6], some of those methods (in particular Re-Pair) perform very well in practice, both in classical and repetitive settings.[3]

On the other hand, unlike LZ77, grammar compression allows one to decompress arbitrary substrings of T almost optimally [14, 4]. The most recent result [4] extracts any $T[p, p+\ell-1]$ in time $O(\ell + \lg u)$, which is optimal. Unfortunately, that representation [4] requires $O(N \lg u)$ bits, possibly proportional but in practice many times the size of the output of a grammar compressor. On the practical side, applications like Comrad [22] achieve good space and time for extracting substrings of T.

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* and *search*, in time depending only polylogarithmically on u. They have appeared in the last decade [28], and have focused mostly on statistical compression. As a result, they work well on classical texts, but not on repetitive collections [24]. Some of those self-indexes have been adapted to repetitive collections [24], but they cannot reach the compression ratio of the best grammar-based methods.

Searching for patterns on grammar-compressed text has been faced mostly in sequential form, that is, scanning the whole grammar. The best result [19] achieves time $O(N + m^2 + occ)$. This may be $o(u)$, but still linear in the size of the compressed text. There exist a few self-indexes based on LZ78-like compression [12, 1, 31], but LZ78 is among the weakest grammar-based compressors. In particular, LZ78 was shown not to be competitive on highly repetitive collections [24].

The only self-index supporting general grammar compressors [10] operates on "straight-line programs" (SLPs), where the right hands of the rules are of length 1 or 2. Given such a grammar they achieve, among other tradeoffs, $3n \lg n + n \lg u$ bits of space and $O(m(m+h) \lg^2 n)$ search time, where h is the height of the parse tree of the grammar. A general grammar of n symbols and size N can be converted into a SLP by adding at most $N - n$ nonterminals and rules, and increasing the height h by up to an $O(\lg(N-n))$ factor.

More recently, a self-index based on LZ77 compression has been developed [21]. Given a parsing of T into $\bar{n}$ phrases, the self-index uses $\bar{n} \lg \bar{n} + 2\bar{n} \lg u + O(\bar{n} \lg \sigma)$ bits of space, and searches in time $O(m^2 \bar{h} + (m + occ) \lg \bar{n})$, where $\bar{h}$ is the nesting of the parsing. Extraction requires $O(\ell \bar{h})$ time, where ℓ is the length of the extracted substring. Experiments on repetitive collections [8, 9] show that the grammar-based compressor [10] can be competitive with the best classical self-index adapted to repetitive collections [24] but, at least that particular implementation, is not competitive with the LZ77-based self-index [21].

Note that the search times in both self-indexes depend on h or $\bar{h}$. This is undesirable as both are only bounded by n or $\bar{n}$. Very recently, a result combining grammars and LZ77 parsing [13] achieved $O(m^2 + (m + occ) \lg \lg \bar{n})$ time and $O(\bar{n} \lg u \lg \lg \bar{n})$ bits (i.e., slightly superlinear on the LZ77 compressed size). Other

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

close variants of LZ77 parsings also yield promising indexing results in particular scenarios [17, 25, 11].

Our main contribution is a new representation of general context-free grammars. The following theorem summarizes its properties. Note that the space is $O(N \lg u)$ and the search time is independent of h.

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

In the rest of the paper we describe how this structure operates. First, we preprocess the grammar to enforce several invariants useful to ensure our time complexities. Then we use a data structure for labeled binary relations [10] to find the "primary" occurrences of P, that is, those formed when concatenating symbols in the right hand of a rule. To get rid of the factor h in this part of the search, we introduce a new technique 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.

2 Basic Concepts

2.1 Sequence Representations

Our data structures use succinct representations of sequences. Given a sequence $S[1..N]$, over alphabet of size n, we need to support the following operations: $access(S, i)$ retrieves the symbol $S[i]$; $rank_a(S, i)$ counts number of occurrences of a in $S[1..i]$; and $select_a(S, j)$ computes position where the jth a appears in S.

For the case $n = 2$, Raman et al. [30] proposed two compressed representations that are useful when the number n' of 1s in S is small. One is the "fully indexable dictionary" (FID). It takes $n' \lg \frac{N}{n'} + O(n' + N \lg \lg N / \lg N)$ bits of space and supports all the operations in constant time. A weaker one is the "indexable dictionary" (ID), that takes $n' \lg \frac{N}{n'} + O(n' + \lg \lg N)$ bits of space and supports in constant time queries $access(S, i)$, $rank(S, i)$ if $S[i] = 1$, and $select_1(S, j)$.

For general sequences, the wavelet tree [16] requires $N \lg n + o(N)$ bits [15] and supports all three operations in $O(\lg n)$ time. Another representation, by Barbay et al. [2], requires $N \lg n + o(N \lg n)$ bits and solves $access(S, i)$ in $O(1)$ time and $select(S, j)$ in time $O(\lg \lg n)$, or vice versa; $rank(S, i)$ takes time $O(\lg \lg n)$.

2.2 Labeled Binary Relations

A labeled binary relation corresponds to a binary relation $\mathcal{R} \subseteq A \times B$, where $A = [1..n_1]$ and $B = [1..n_2]$, augmented with a function $\mathcal{L} : A \times B \rightarrow L \cup \{\perp\}$, $L = [1..\ell]$, 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. We describe a simplification of a representation of binary relations [10], for the case of this paper where each element of A is associated to exactly one element of B, so $|\mathcal{R}| = n_1$. We use a string $S_B[1..n_1]$ over alphabet $[1..n_2]$, where $S_B[i]$ is the element of B associated to column i. A second string $S_\mathcal{L}[1..n_1]$ on alphabet $[1..\ell]$ is stored, so that $S_\mathcal{L}[i]$ is the label of the pair represented by $S_B[i]$.

If we use a wavelet tree for S_B and a plain string representation for $S_\mathcal{L}$, the total space is $n_1(\lg n_2 + \lg \ell) + O(n_1)$ bits. With this representation we can answer, among others, the following queries of interest in this paper: (1) Find the label of the element b associated to 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 $O((k + 1) \lg n_2)$ time.

2.3 Succinct Tree Representations

There are many tree representations for trees $\mathcal{T}$ with N nodes that take $2N + o(N)$ bits of space. In this paper we use one called DFUDS [3], which in particular answers in constant time the following operations (node identifiers v are associated to a position in $[1..2N]$): $node_\mathcal{T}(p)$ is the node with preorder number p; $preorder_\mathcal{T}(v)$ is the preorder number of node v; $leafrank_\mathcal{T}(v)$ is the number of leaves to the left of v; $numleaves_\mathcal{T}(v)$ is the number of leaves below v; $parent_\mathcal{T}(v)$ is the parent of v; $child_\mathcal{T}(v, k)$ is the kth child of v; $nextsibling_\mathcal{T}(v)$ is the next sibling of v; $rchild_\mathcal{T}(v)$ is k such that v is the kth child of its parent; $degree_\mathcal{T}(v)$ is the number of children of v; $depth_\mathcal{T}(v)$ is the depth of v; and $level\text{-}ancestor_\mathcal{T}(v, k)$ is the kth ancestor of v.

The DFUDS representation is obtained by traversing the tree in DFS order and appending to a bitmap the degree of each node, written in unary.

3 Preprocessing and Representing the Grammar

Let $\mathcal{G}$ be a grammar that generates a single string $T[1..u]$, formed by n (terminal and nonterminal) symbols. The $\sigma \leq n$ terminal symbols come from an alphabet $\Sigma = [1, \sigma]$,[4] and then $\mathcal{G}$ contains $n - \sigma$ rules of the form $X_i \rightarrow \alpha_i$, one per nonterminal. This α_i is called the *right-hand side* of the rule, and corresponds to the sequence of terminal and non-terminal symbols generated by X_i (without recursively unrolling rules). We call $N = \sum |\alpha_i|$ the *size* of $\mathcal{G}$. Note it holds $\sigma \leq N$, as the terminals must appear in the right-hand sides. We assume all the nonterminals are used to generate the string; otherwise unused rules can be found and dropped in $O(N)$ time.

[4] Non-contiguous alphabets can be handled with some extra space [10].

We preprocess $\mathcal{G}$ as follows. First, for each terminal symbol $a \in \Sigma$ present in $\mathcal{G}$ we create a rule $X_a \to a$, and replace all other occurrences of a in the grammar by X_a. As a result, the grammar contains exactly n nonterminal symbols $\mathcal{X} = \{X_1, \ldots, X_n\}$, each associated to a rule $X_i \to \alpha_i$, where $\alpha_i \in \Sigma$ or α_i is a sequence of elements in X. We assume that X_n is the start symbol.

Any rule $X_i \to \alpha_i$ where $|\alpha_i| \leq 1$ (except for $X_a \to a$) is removed by replacing X_i by α_i everywhere, decreasing n and without increasing N.

We further preprocess $\mathcal{G}$ to enforce the property that any nonterminal X_i, except X_n and those $X_i \to a \in \Sigma$, 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 rewritten, and as we rewrite a right-hand side, we replace any (non-excepted) X_i that appears once by its right-hand side α_i. This transformation takes $O(N)$ time and does not alter N (yet it may reduce n).

Note n is now the number of rules in the transformed grammar $\mathcal{G}$. We will still call N the size of the original grammar (the transformed one has size $\leq N + \sigma$; similarly its number of rules is at most $n + \sigma$).

We call $\mathcal{F}(X_i)$ the single string generated by X_i, that is $\mathcal{F}(X_i) = a$ if $X_i \to a$ and $\mathcal{F}(X_i) = \mathcal{F}(X_{i_1}) \ldots \mathcal{F}(X_{i_k})$ if $X_i \to X_{i_1} \ldots X_{i_k}$. $\mathcal{G}$ generates the text $T = \mathcal{L}(\mathcal{G}) = \mathcal{F}(X_n)$.

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 S^{rev} is string S read backwards (usefulness of this will be apparent later). The sorting can be done in time $O(u + n \lg n)$ and $O(u \lg u)$ bits of space [10], which dominates the previous time complexities. Let us say that X_n became X_s after the reordering.

We define now a structure that will be key in our index.

Definition 1. *The* grammar tree *of $\mathcal{G}$ is a general tree $\mathcal{T}_{\mathcal{G}}$ with nodes labeled in $\mathcal{X}$. Its root is labeled X_s. Let $\alpha_s = X_{s_1} \ldots X_{s_k}$. Then the root has k children labeled $X_{s_1}, \ldots, X_{s_k}$. The subtrees of these children are defined recursively, left to right, so that the first time we find a symbol X_i in the parse tree, we define its children using α_i. However, the next times we find a symbol X_i in our recursive left-to-right traversal, we leave it as a leaf of the grammar tree (if we expanded it, the resulting tree would be the* parse tree *of T with u leaf nodes). Also symbols $X_a \to a$ are not expanded but left as leaves. We say that X_i is* defined *in the only internal node of $\mathcal{T}_{\mathcal{G}}$ labeled X_i.*

Since each right-hand side $\alpha_i \neq a \in \Sigma$ is written once in the tree, plus the root X_s, the total number of nodes in $\mathcal{T}_{\mathcal{G}}$ is $N + 1$.

The grammar tree partitions T in a way that is useful for finding occurrences, using a concept that dates back to Kärkkäinen [18].

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 $T = T_1 T_2 \ldots$ is a partition of T according to the leaves of $\mathcal{T}_{\mathcal{G}}$. We call occurrences of a pattern P* primary *if they span more than one T_i. Other occurrences are called* secondary.

Our self-index will represent $\mathcal{G}$ using two main components. One represents the grammar tree $\mathcal{T}_{\mathcal{G}}$ using a DFUDS representation (Sec. 2.3) and a sequence of labels (Sec. 2.1). This will be used to extract text and decompress rules. When augmented with a secondary trie $\mathcal{T}_S$ storing leftmost/rightmost paths in $\mathcal{T}_{\mathcal{G}}$, the representation will expand any prefix/suffix of a rule in optimal time [14].

The second component in our self-index corresponds to a labeled binary relation (Sec. 2.2), where $B = \mathcal{X}$ and A is the set of proper suffixes starting at positions $j + 1$ of rules α_i: $(\alpha_i[j], \alpha_i[j + 1..])$ will be related for all $X_i \rightarrow \alpha_i$ and $1 \leq j < |\alpha_i|$. The labels are numbers in the range $[1, N + 1]$; we specify their meaning later. This binary relation will be used to find the primary occurrences of the search pattern. Secondary occurrences will be tracked in the grammar tree.

4 Extracting Text

We first describe a simple structure that extracts the prefix of length ℓ of any rule in $O(\ell + h)$ time. We then augment this structure to support extracting any substring of length ℓ in time $O(\ell + h \lg(N/h))$, and finally augment it further to retrieve the prefix or suffix of any rule in optimal $O(\ell)$ time. This last result is fundamental for supporting searches, and is obtained by extending the structure proposed by Gasieniec et al. [14] for SLPs to general context-free grammars generating one string. The improvement does not work for extracting arbitrary substrings, as in that case one has to find first the nonterminals that must be expanded. This subproblem is not easy, especially in little space [4].

As said, we represent the topology of the grammar tree $\mathcal{T}_{\mathcal{G}}$ using DFUDS [3]. The sequence of labels associated to the tree nodes is stored in preorder in a sequence $X[1..N + 1]$, using the fast representation of Sec. 2.1 where we choose constant time for $access(X, i) = X[i]$ and $O(\lg \lg n)$ time for $select_a(X, j)$.

We also store a bitmap $Y[1..n]$ that marks the rules of the form $X_i \rightarrow a \in \Sigma$ with a 1-bit. Since the rules have been renumbered in (reverse) lexicographic order, every time we find a rule X_i such that $Y[i] = 1$, we can determine the terminal symbol it represents as $a = rank_1(Y, i)$ in constant time.

4.1 Expanding Prefixes of 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 rest are leaves in $\mathcal{T}_{\mathcal{G}}$. Therefore, $v = node_{\mathcal{T}_{\mathcal{G}}}(select_{X_i}(X, 1))$ is the node in $\mathcal{T}_{\mathcal{G}}$ where X_i is defined. We traverse the subtree rooted at v in DFS order. Every time we reach a leaf u, we compute its label $X_j = X[preorder_{\mathcal{T}_{\mathcal{G}}}(u)]$, and either output a terminal if $Y[j] = 1$ or recursively expand X_j. This is in fact a traversal of the *parse tree* starting at node v, using instead the grammar tree. Such a traversal takes $O(\ell + h_v)$ steps [10], where $h_v \leq h$ is the height of the parsing subtree rooted at v. In particular, if we extract the whole rule X_i we pay $O(\ell)$ steps, since we have removed unary paths in the preprocessing of $\mathcal{G}$ and thus v has $\ell > h_v$ leaves in the parse tree.

The only obstacle to having constant-time steps are the queries $select_{X_i}(X, 1)$. As these are only for the position 1, we can have them precomputed in a sequence $F[1..n]$ using $n\lceil \lg N\rceil = n \lg n + O(N)$ further bits of space.

The total space required for $\mathcal{T}_\mathcal{G}$, considering the DFUDS topology, sequence X, bitmap Y, and sequence F, is $N \lg n + n \lg n + o(N \lg n)$ bits. We reduce the space to $N \lg n + \delta\, n \lg n + o(N \lg n)$, for any $0 < \delta \leq 1$, as follows. Form a sequence $X'[1..N - n + 1]$ where the first position of every symbol X_i in X has been removed, and mark in a bitmap $Z[1..N + 1]$, with a 1, those first positions in X. Replace our sequence F by a permutation $\pi[1..n]$ so that $select_{X_i}(X, 1) = F[i] = select_1(Z, \pi[i])$. Now we can still access any $X[i] = X'[rank_0(Z, i)]$ if $Z[i] = 0$. For the case $Z[i] = 1$ we have $X[i] = \pi^{-1}[rank_1(Z, i)]$. Similarly, $select_{X_i}(X, j) = select_0(Z, select_{X_i}(X', j - 1))$ for $j > 1$. Then use Z, π, and X' instead of F and X.

All the operations retain the same times except for the access to π^{-1}. We use for π a representation by Munro et al. [27] that takes $(1 + \delta)n \lg n$ bits and computes any $\pi[i]$ in constant time and any $\pi^{-1}[j]$ in time $O(1/\delta)$, which will be the cost to access X. Although this will have an impact later, we note that for extraction we only access X at leaf nodes, where it always takes constant time.[5]

4.2 Extracting Arbitrary Substrings

In order to extract any given substring of T, we add a bitmap $L[1..u + 1]$ that marks with a 1 the first position of each T_i in T. We can then compute the starting position of any node $v \in \mathcal{T}_\mathcal{G}$ as $select_1(L, leafrank_{\mathcal{T}_\mathcal{G}}(v) + 1)$.

To extract $T[p, p + \ell - 1]$, we binary search the starting position p from the root of $\mathcal{T}_\mathcal{G}$. If we arrive at a leaf not representing a terminal, we go to its definition in $\mathcal{T}_\mathcal{G}$, translate position p to the area below the new node v, and continue recursively. At some point we finally reach position p, and from there on we extract the symbols rightwards. Just as before, the total number of steps is $O(\ell + h)$. Yet, the h steps require binary searches. As there are at most h binary searches among the children of different tree nodes, and there are $N + 1$ nodes, at worst the binary searches cost $O(h \lg(N/h))$. The total cost is $O(\ell + h \lg(N/h))$.

The number of ones in L is at most N. Since we only need $select_1$ on L, we can use an ID representation (see Sec. 2.1), requiring $N \lg(u/N) + O(N + \lg\lg u) = N \lg(u/N) + O(N)$ bits (since $N \geq \lg u$ in any grammar). Thus the total space becomes $N \lg n + N \lg(u/N) + \delta\, n \lg n + o(N \lg n)$ bits.

4.3 Optimal Expansion of Rule Prefixes and Suffixes

Our improved version builds on the proposal by Gasieniec et al. [14]. We extend their representation using succinct data structures in order to handle general

[5] Nonterminals $X_a \rightarrow a$ do not have a definition in $\mathcal{T}_\mathcal{G}$, so they are not extracted from X nor represented in π, thus they are accessed in constant time. They can be skipped from $\pi[1..n]$ with bitmap Y, so that in fact π is of length $n - \sigma$ and is accessed as $\pi[rank_0(Y, i)]$; for π^{-1} we actually use $select_0(Y, \pi^{-1}[j])$.

grammars instead of only SLPs. 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 $X_a \in \mathcal{X}$, not the terminals $a \in \Sigma$). We insert all the strings $S(X_i)^{rev}$ into a trie $\mathcal{T}_S$. Note that each symbol appears only once in $\mathcal{T}_S$ [14], thus it has n nodes. Again, we represent the topology of $\mathcal{T}_S$ using DFUDS. Yet, its sequence of labels $X_S[1..n]$ turns out to be a permutation in $[1..n]$, for which we use again the structure [27] that takes $(1+\epsilon)n \lg n$ bits and computes any $X_S[i]$ in constant time and any $X_S^{-1}[j]$ in time $O(1/\epsilon)$.

We can 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 representing some $a \in \Sigma$, it follows that X_i descends in $\mathcal{T}_S$ from X_a, which is a child of the root. This node is $v_a = \mathit{level\text{-}ancestor}_{\mathcal{T}_S}(v, \mathit{depth}_{\mathcal{T}_S}(v) - 1)$. Then $a = \mathit{rank}_1(Y, X_S[\mathit{preorder}_{\mathcal{T}_S}(v_a)])$.

A prefix of X_i is extracted as follows. First, we obtain the corresponding node $v \in \mathcal{T}_S$ as $v = X_S^{-1}[X_i]$. Then we obtain the leftmost symbol of v as explained. 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, of the nodes in the downward path from the root of $\mathcal{T}_S$ to v. Therefore, for each node w in the list $\mathit{level\text{-}ancestor}_{\mathcal{T}_S}(v, \mathit{depth}_{\mathcal{T}_S}(v) - 2), \ldots, \mathit{parent}_{\mathcal{T}_S}(v), v$, we map w to its definition $x \in \mathcal{T}_\mathcal{G}$, $x = \mathit{node}_{\mathcal{T}_\mathcal{G}}(\mathit{select}_{X_j}(X, 1))$ where $X_j = X_S[\mathit{preorder}_{\mathcal{T}_S}(w)]$. Once x is found, we recursively expand its children, from the second onwards, by mapping them back to $\mathcal{T}_S$, and so on. Charging the cost to the new symbol to expand, and since there are no unary paths, it follows that we carry out $O(\ell)$ steps to extract the first ℓ symbols, and the extraction is real-time [14]. All costs per step are $O(1)$ except for the $O(1/\epsilon)$ to access X_S^{-1}.

For extracting suffixes of rules in $\mathcal{G}$, we need another version of $\mathcal{T}_S$ that stores the rightmost paths. This leads to our first result (choosing $\delta = o(1)$).

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

5 Locating Patterns

A secondary occurrence of the pattern P inside a leaf of $\mathcal{T}_\mathcal{G}$ labeled by a symbol X_i occurs as well in the internal node of $\mathcal{T}_G$ where X_i is defined. If that occurrence is also secondary, then it occurs inside a child X_j of X_i, and we can repeat the argument with X_j until finding a primary occurrence inside some X_k. Thus, to find all the secondary occurrences, we can first spot the primary occurrences, and then find all the copies of the nonterminal X_k that contain the primary occurrences, as well as all the copies of the nonterminals that contain X_k, recursively.

We base our approach on the strategy proposed by Kärkkäinen [18] to find the primary occurrences of $P = p_1 p_2 \ldots p_m$. Kärkkäinen considers the $m-1$

partitions $P = P_1 \cdot P_2$, $P_1 = p_1 \dots p_i$ and $P_2 = p_{i+1} \dots p_m$, for $1 \leq i < m$. In our case, for each partition we will find all the nonterminals $X_k \to X_{k_1} X_{k_2} \dots X_{k_r}$ such that P_1 is a suffix of some $\mathcal{F}(X_{k_i})$ and P_2 is a prefix of $\mathcal{F}(X_{k_{i+1}}) \dots \mathcal{F}(X_{k_r})$. This finds each primary occurrence exactly once. The secondary occurrences are then tracked in the grammar tree $\mathcal{T}_\mathcal{G}$. We handle the case $m = 1$ by finding all occurrences of X_{p_1} in $\mathcal{T}_\mathcal{G}$ using *select* over the labels, and treat them as primary occurrences.

5.1 Finding Primary Occurrences

As anticipated at the end of Sec. 3, we store a binary relation $\mathcal{R} \subseteq A \times B$ to find the primary occurrences. It has n rows labeled X_i, for all $X_i \in \mathcal{X} = B$, and $N - n$ columns[6]. Each column corresponds to a distinct proper suffix $\alpha_i[j+1..]$ of a right-hand side α_i. The labels belong to $[1..N+1]$. The relation contains one pair per column: $(\alpha_i[j], \alpha_i[j+1..]) \in \mathcal{R}$ for all $1 \leq i \leq n$ and $1 \leq j < |\alpha_i|$. Its label is the preorder of the $(j+1)$th child of the node that defines X_i in $\mathcal{T}_\mathcal{G}$. The space for the binary relation is $(N-n)(\lg n + \lg N) + O(N)$ bits.

Recall that, in our preprocessing, we have sorted $\mathcal{X}$ according to the lexicographic order of $\mathcal{F}(X_i)^{rev}$. We also sort the suffixes $\alpha_i[j+1..]$ lexicographically with respect to their expansion, that is, $\mathcal{F}(\alpha_i[j+1])\mathcal{F}(\alpha_i[j+2]) \dots \mathcal{F}(\alpha_i[|\alpha_i|])$. This can be done in $O(u + N \lg N)$ time in a way similar to how $\mathcal{X}$ was sorted: Each suffix $\alpha_i[j+1..]$, labeled p, can be associated to the substring $T[select_1(L, rankleaf_{\mathcal{T}_\mathcal{G}}(node_{\mathcal{T}_\mathcal{G}}(p)) + 1) \dots select_1(L, rankleaf_{\mathcal{T}_\mathcal{G}}(v) + 1 + numleaves_{\mathcal{T}_\mathcal{G}}(v)) - 1]$, where v is the parent of $node_{\mathcal{T}_\mathcal{G}}(p)$. Then we can proceed as in previous work [10].

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 $\mathcal{F}(X_i)^{rev}$. Each comparison in the binary search needs to extract $|P_1|$ terminals from the suffix of $\mathcal{F}(X_i)$. According to Lemma 1, this takes $O(|P_1|/\epsilon)$ time. Similarly, we binary search for the range of columns whose expansions start with P_2. Each comparison needs to extract $\ell = |P_2|$ terminals from the prefix of $\mathcal{F}(\alpha_i[j+1])\mathcal{F}(\alpha_i[j+2]) \dots$. Let r be the column we wish to compare to P_2. We extract the label p associated to the column in constant time (recall Sec. 2.2). Then we extract the first ℓ symbols from the expansion of $node_{\mathcal{T}_\mathcal{G}}(p)$. If $node_{\mathcal{T}_\mathcal{G}}(p)$ does not have enough symbols, we continue with $nextsibling_{\mathcal{T}_\mathcal{G}}(p)$, and so on, until we extract ℓ symbols or we exhaust the suffix of the rule. According to Lemma 1, this requires time $O(|P_2|/\epsilon)$. Thus our two binary searches require time $O((m/\epsilon) \lg N)$.

This time can be further improved by using the same technique as in previous work [10]. The idea is to sample phrases at regular intervals and store the sampled phrases in a Patricia tree [26]. We first search for the pattern in the Patricia tree, and then complete the process with a binary search between two sampled phrases (we first verify the correctness of the Patricia search by checking that our pattern is actually within the range found). By sampling every $\lg u \lg\lg n / \lg n$ phrases,

[6] Recall $\mathcal{F}(X_i) \leq \mathcal{F}(X_j)$ iff $i \leq j$.

the resulting time for searching becomes $O\left(m \lg\left(\frac{\lg u}{\lg n}\right)\right)$ and we only require $o(N \lg n)$ bits of extra space, as the Patricia tree needs $O(\lg u)$ bits per node.

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 $O((k+1)\lg n)$, according to Sec. 2.2. The parents of all the nodes $node_{\mathcal{T}_\mathcal{G}}(p)$, for each point p in the range, correspond to the primary occurrences. In Sec. 5.2 we show how to report primary and secondary occurrences starting directly from those $node_{\mathcal{T}_\mathcal{G}}(p)$ positions.

We have to carry out this search for $m-1$ partitions of P, whereas each primary occurrence is found exactly once. Calling occ the number of primary occurrences, the total cost of this part of the search is $O\left((m^2/\epsilon)\lg\left(\frac{\lg u}{\lg n}\right) + (m + occ)\lg n\right)$.

5.2 Tracking 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 these occur in T. Given a primary occurrence for partition $P = P_1 \cdot P_2$ located at $u = node_{\mathcal{T}_\mathcal{G}}(p)$, we obtain the starting position of P in T by moving towards the root while keeping count of the offset between the beginning of the current node and the occurrence of P. Initially, for node u itself, this is $l = -|P_1|$. Now, while u is not the root, we set $l \leftarrow l + select_1(L, rankleaves_{\mathcal{T}_\mathcal{G}}(u)+1) - select_1(L, rankleaves_{\mathcal{T}_\mathcal{G}}(parent_{\mathcal{T}_\mathcal{G}}(u))+1)$, then $u \leftarrow parent_{\mathcal{T}_\mathcal{G}}(u)$. When we reach the root, the occurrence of P starts at l.

It seems like we are doing this h times in the worst case, since we need to track the occurrence up to the root. In fact we might do so for some symbols, but the total cost is amortized. Every time we move from u to $v = parent_{\mathcal{T}_\mathcal{G}}(u)$, we know that $X[v]$ appears at least once more in the tree. This is because of our preprocessing (Sec. 3), where we force rules to appear at least twice or be removed. Thus v defines $X[v]$, but there are one or more leaves labeled $X[v]$, and we have to report the occurrences of P inside them all. For this sake we carry out $select_{X[v]}(X, i)$ for $i = 1, 2\ldots$ until spotting all those occurrences (where P occurs with the current offset l). We recursively track them to the root of $\mathcal{T}_\mathcal{G}$ to find their absolute position in T, and recursively find the other occurrences of all their ancestor nodes. The overall cost amortizes to $O(1)$ steps per occurrence reported, as we can charge the cost of moving from u to v to the other occurrence of v. If we report occ secondary occurrences we carry out $O(occ)$ steps, each costing $O(\lg\lg n)$ time. We can thus use $\delta = O(1/\lg\lg n)$ (Sec. 4.1) so that the cost to access $X[v]$ does not impact the space nor time complexity.

By adding up the space of Lemma 1 with that of the labeled binary relation, and adding up the costs, we have our central result, Theorem 1.

6 Conclusions

We presented the first grammar-based text index with locating time independent on the height of the grammar. There are previous results on generating balanced grammars to compress text, as the ones proposed by Rytter [32] and Sakamoto

[33]. These representations allow previous indexing techniques to guarantee sublinear locating times, yet they introduce a penalty in the size of the grammar. Our index also extends the grammar-based indexing techniques to a more general class of grammars than SLPs, the only class explored so far in this scenario.

Our extraction time, $O(\ell+h\lg(N/h))$, is not the optimal $O(\ell+\lg u)$. This time can be achieved by adding $O(N\lg u)$ bits [4]. Within this space, our search time can be improved by, in Sec. 5.1, (1) using full (not sampled) Patricia trees, and (2) using a faster grid representation [5] to speed up primary occurrences (secondary ones already take $O(occ\lg\lg n)$ time). This yields the following simplified result:

Corollary 1. *Let a sequence $T[1..u]$ be represented by a context free grammar of size N, and let $0 < \epsilon < 1$ be any constant. Then, there exists a data structure using $O(N\lg u)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O(m^2+(m+occ)\lg^\epsilon N)$. It can extract any substring of length ℓ from T in time $O(\ell+\lg u)$.*

Several questions remain open: Is it possible to lower the dependence on m to linear, as in some more limited schemes [11, 25, 17, 31]? Is it possible to reduce the space to $N\lg n + o(N\lg n)$, that is, asymptotically the same as the compressed text, as on statistical-compression-based self-indexes [28]? Is it possible to remove h from the extraction complexity within less space than the current solutions [4]?

References

1. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proc. 17th CPM. pp. 319–330 (2006)
2. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Proc. 21st ISAAC. pp. 315–326 (part II) (2010)
3. Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: Proc. 22nd SODA. pp. 373–389 (2011)
5. Chan, T., Larsen, K., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. 27th SoCG. pp. 1–10 (2011)
6. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theo. 51(7), 2554–2576 (2005)
7. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Rasala, A., Sahai, A., Shelat, A.: Approximating the smallest grammar: Kolmogorov complexity in natural models. In: STOC. pp. 792–801 (2002)
8. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Compressed q-gram indexing for highly repetitive biological sequences. In: Proc. 10th BIBE (2010)
9. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Indexes for highly repetitive document collections. In: Proc. 20th CIKM. pp. 463–468 (2011)
10. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fund. Inf. 111(3), 313–337 (2010)

11. Do, H.H., Jansson, J., Sadakane, K., Sung, W.K.: Fast relative lempel-ziv self-index for similar sequences. In: Proc. FAW-AAIM. pp. 291–302 (2012)
12. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM 52(4), 552–581 (2005)
13. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.: A faster grammar-based self-index. In: Proc. 6th LATA, pp. 240–251 (2012)
14. Gasieniec, L., Kolpakov, R., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proc. 15th DCC. pp. 458–458 (2005)
15. Golynski, A., Raman, R., Rao, S.: On the redundancy of succinct data structures. In: Proc. 11th SWAT. pp. 148–159. LNCS 5124 (2008)
16. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
17. Huang, S., Lam, T.W., Sung, W.K., Tam, S.L., Yiu, S.M.: Indexing similar DNA sequences. In: Proc. 6th AAIM. pp. 180–190 (2010)
18. Kärkkäinen, J.: Repetition-Based Text Indexing. Ph.D. thesis, Department of Computer Science, University of Helsinki, Finland (1999)
19. Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: a unifying framework for compressed pattern matching. Theor. Comp. Sci. 298(1), 253–272 (2003)
20. Kieffer, J., Yang, E.H.: Grammar-based codes: A new class of universal lossless source codes. IEEE Trans. Inf. Theo. 46(3), 737–754 (2000)
21. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Proc. 22th CPM. pp. 41–54 (2011)
22. Kuruppu, S., Beresford-Smith, B., Conway, T., Zobel, J.: Repetition-based compression of large DNA datasets. In: Proc. 13th RECOMB (2009), poster
23. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
24. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Proc. 13th RECOMB. pp. 121–137 (2009)
25. Maruyama, S., Nakahara, M., Kishiue, N., Sakamoto, H.: ESP-index: A compressed index based on edit-sensitive parsing. In: Proc. 18th SPIRE. pp. 398–409 (2011)
26. Morrison, D.: PATRICIA – practical algorithm to retrieve information coded in alphanumeric. J. ACM 15(4), 514–534 (1968)
27. Munro, J., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Proc. 30th ICALP. pp. 345–356 (2003)
28. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
29. Nevill-Manning, C., Witten, I., Maulsby, D.: Compression by induction of hierarchical grammars. In: Proc. 4th DCC. pp. 244–253 (1994)
30. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA. pp. 233–242 (2002)
31. Russo, L., Oliveira, A.: A compressed self-index using a Ziv-Lempel dictionary. Inf. Ret. 11(4), 359–388 (2008)
32. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theo. Comp. Sci. 302(1-3), 211–222 (2003)
33. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discr. Alg. 3, 416–430 (2005)
34. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theo. 23(3), 337–343 (1977)
35. Ziv, J., Lempel, A.: Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theo. 24(5), 530–536 (1978)