Fundamenta Informaticae XXI (2001) 1001–1023 1001
IOS Press

Self-Indexed Grammar-Based Compression *

Francisco Claude†
David R. Cheriton School of Computer Science
University of Waterloo
fclaude@cs.uwaterloo.ca

Gonzalo Navarro‡
Department of Computer Science
University of Chile
gnavarro@dcc.uchile.cl

Abstract. Straight-line programs (SLPs) offer powerful text compression by representing a text $T[1,u]$ in terms of a restricted context-free grammar of n rules, so that T can be recovered in $O(u)$ time. However, the problem of operating the grammar in compressed form has not been studied much. We present a grammar representation whose size is of the same order of that of a plain SLP representation, and can answer other queries apart from expanding nonterminals. This can be of independent interest. We then extend it to achieve the first grammar representation able of extracting text substrings, and of searching the text for patterns, in time $o(n)$. We also give byproducts on representing binary relations.

Keywords: Grammar-based Compression, Pattern Matching, Data Structures.

1. Introduction and Related Work

Grammar-based compression is a well-known technique since at least the seventies, and still a very active area of research. From the different variants of the idea, we focus on the case where a given text $T[1,u]$

Address for correspondence: Address for correspondence goes here
*A partial early version of this paper appeared in *Proc. MFCS 2009, pp. 235–246.*
†Funded in part by NSERC Canada and Go-Bell Scholarships program.
‡Funded in part by Millennium Institute on Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

is replaced by a context-free grammar (CFG) $\mathcal{G}$ that generates just the string T. Then one can store $\mathcal{G}$ instead of T, and this has shown to provide a universal compression method [22]. Some examples are LZ78 [41], Re-Pair [24] and Sequitur [33], among many others [7].

When a CFG deriving a single string is converted into Chomsky Normal Form, the result is essentially a *Straight-Line Program (SLP)*, that is, a grammar where each nonterminal appears once at the left-hand side of a rule, and can either be converted into a terminal or into the concatenation of two previous nonterminals. SLPs are thus as powerful as CFGs for our purpose, and the grammar-based compression methods above can be straightforwardly translated, with no significant penalty, into SLPs. SLPs are in practice competitive with the best compression methods [13].

There are textual substitution compression methods which are more powerful than those CFG-based [21]. A well-known one is LZ77 [40], which cannot be directly expressed using CFGs. Yet, an LZ77 parsing can be converted into an SLP with an $O(\log u)$ penalty factor in the size of the grammar, which might be preferable as SLPs are much simpler to manipulate [37].

SLPs have received attention because, despite their simplicity, they are able to capture the redundancy of highly repetitive strings. Indeed, an SLP of n rules can represent a text exponentially longer than n. They are also attractive because decompression is easily carried out in linear time. Compression, instead, is more troublesome. Finding the smallest SLP that represents a given text $T[1,u]$ is NP-complete [37, 7]. Moreover, some popular grammar-based compressors such as LZ78, Re-Pair and Sequitur, can generate a compressed file much larger than the smallest SLP [7]. Yet, a simple method to achieve an $O(\log u)$-approximation is to parse T using LZ77 and then converting it into an SLP [37], which in addition is *balanced*: the height of the derivation tree for T is $O(\log u)$. (Also, any SLP can be balanced by paying an $O(\log u)$ space penalty factor.)

Compression is regarded nowadays not just as an aid for cheap archival or transmission. Since the last decade, the concept of *compressed text databases* has gained momentum. The idea is to handle a large text collection in compressed form all the time, and decompress just for displaying. Compressed text databases require at least two basic operations over a text $T[1,u]$: *extract* and *find*. Operation *extract* returns any desired portion $T[l,l+m]$ of the text. Operation *find* returns the positions of T where a given search pattern $P[1,m]$ occurs in T. We refer as occ to the number of occurrences returned by a *find* operation. *Extract* and *find* should be carried out in $o(u)$ time in order to be practical for large databases.

There has been some work on random access to grammar-based compressed text, without decompressing all of it [12]. As for finding patterns, there has been much work on *sequential* compressed pattern matching [1], that is, scanning the whole grammar. The most attractive result is that of Kida et al. [21], which can search general SLPs/CFGs in time $O(n+m^2+occ)$. This may be $o(u)$, but still linear in the size of the compressed text. A more ambitious goal is *indexed* searching, where data structures are built on the compressed text to permit searching in $o(n)$ time (at least for small enough m and occ).

Indeed, there has been much work on implementing compressed text databases supporting the operations *extract* and *find* efficiently (usually in $O(m\,\mathrm{polylog}(u))$ time) [32], but generally based on the Burrows-Wheeler Transform or Compressed Suffix Arrays, not on grammar compression. The only exceptions are based on LZ78-like compression [31, 10, 36]. These are *self-indexes*, meaning that the compressed text representation itself can support indexed searches. The fact that no (or weak) grammar compression is used makes these self-indexes not sufficiently powerful to cope with highly repetitive text collections, which arise in applications such as computational biology, software repositories, transaction logs, versioned documents, temporal databases, etc. This type of applications require self-indexes based on stronger compression methods, such as general SLPs.

As an example, a recent study modeling a genomics application [39] concluded that none of the existing self-indexes was able to capture the redundancies present in the collection. Even the LZ78-based ones failed, which is not surprising given that LZ78 can output a text exponentially larger than the smallest SLP. The scenario [39] considers a set of r genomes of length u, of individuals of the same species, and can be modeled as r copies of a *base sequence*, where s *edit operations* (substitutions, to simplify) are randomly placed. The most compact self-indexes [32, 15, 11] occupy essentially urH_k bits, where H_k is the k-th order entropy of the base sequence, but this is multiplied r times because they are unable of exploiting long-range repetitions. The powerful LZ77, instead, is able to achieve $uH_k + O((r+s)\log u)$ bits, that is, the compressed base sequence plus $O(\log u)$ bits per edit and per sequence. However, self-indexes based on LZ77 are extremely challenging and do not exist yet. SLPs are able to achieve a result closer to LZ77 than to the current self-indexes in terms of compression and, as we show in this paper, offer self-indexing capabilities.

In this paper we introduce the *first* SLP representation that can support operations *extract* and *find* in $o(n)$ time. More precisely, a plain SLP representation takes $2n\log n$ bits[1], as each new rule expands into two other rules. Our representation takes $O(n\log n)+n\log u$ bits. It can carry out *extract* in time $O((m+h)\log n)$, where h is the height of the derivation tree, and *find* in time $O((m(m+h)+h\,occ)\log n)$ (see the detailed results in Theorem 5.1). A part of our index is a representation for SLPs which takes $2n\log n(1+o(1))$ bits and is able of retrieving any rule in time $O(\log n)$, but also of answering other queries on the grammar within the same time, such as finding the rules mentioning a given non-terminal. We also show how to represent a labeled binary relation, and in addition support a kind of range query.

Our result constitutes a self-index building on much stronger compression methods than the existing ones, and as such, it has the potential of being extremely useful to implement compressed text databases, in particular the very repetitive ones, by combining good compression and efficient indexed searching. Our method is independent on the way the SLP is generated, and as such it can be coupled with different SLP construction algorithms, which might fit different applications. We give some preliminary experimental results that display the potential of the method.

The paper is organized as follows. In Section 2 we give the needed basic concepts to follow the paper. In Section 3 we extend an existing representation of binary relations, so as to support labels and range queries. Section 4 builds on this result to introduce a representation for SLPs that occupies asymptotically the same space of a plain representation, yet it answers a number of useful queries on the grammar in logarithmic time. This is used in Section 5 as a building block for a self-index based on SLPs, supporting substring extraction and pattern searches. In Section 6 we specialize our general result to a couple of well-known grammar compressors, such as Re-Pair and LZ78, and give some preliminary experimental results. Section 7 concludes and gives several open problems and lines for future research.

2. Basic Concepts

2.1. Rank/Select Data Structures

We make heavy use of succinct data structures for representing sequences with support for *rank/select* and for range queries.

Given a sequence S of length n, drawn from an alphabet Σ of size σ:

[1] In this paper log stands for $\log_2$ unless stated otherwise.

- $rank_S(a, i)$ counts the occurrences of symbol $a \in \Sigma$ in $S[1, i]$, $rank_S(a, 0) = 0$.
- $select_S(a, i)$ finds the i-th occurrence of symbol $a \in \Sigma$ in S, $select_S(a, 0) = 0$.

We also require that data structures representing S provide operation $access_S(i) = S[i]$.

For the special case $\Sigma = \{0, 1\}$, the problem has been solved using $n + o(n)$ bits of space while answering the three queries in constant time [8]. This was later improved to use $O(m \log \frac{n}{m}) + o(n)$ bits, where m is the number of bits set in the bitmap [34].

The general case has been a little harder. Wavelet trees [15] achieve $n \log \sigma + o(n) \log \sigma$ bits of space while answering all the queries in $O(\log \sigma)$ time. This was later improved [11] with multiary wavelet trees to achieve $O(1 + \frac{\log \sigma}{\log \log n})$ time within the same space. Another interesting proposal [14], focused on large alphabets, achieves $n \log \sigma + n\, o(\log \sigma)$ bits of space and answers *rank* and *access* in $O(\log \log \sigma)$ time, while *select* takes $O(1)$ time. Another tradeoff within the same space [14] is $O(1)$ time for *access*, $O(\log \log \sigma)$ time for *select*, and $O(\log \log \sigma \log \log \log \sigma)$ time for *rank*.

2.2. Range Queries on Wavelet Trees

The wavelet tree reduces the *rank/select/access* problem for general alphabets to those on binary sequences. It is a perfectly balanced tree that stores a bitmap of length n at the root; every position in the bitmap is either 0 or 1 depending on whether the symbol at this position belongs to the first half of the alphabet or to the second. The left child of the root will handle the subsequence of S marked with a 0 at the root, and the right child will handle the 1s. This decomposition into alphabet subranges continues recursively until reaching level $\lceil \log \sigma \rceil$, where the leaves correspond to individual symbols.

Mäkinen and Navarro [25] showed how to use a wavelet tree to represent a permutation π of $[1, n]$ so as to answer *range queries*. We give here a slight extension we use in this paper. Given a general sequence $S[1, n]$ over alphabet $[1, \sigma]$, we use the wavelet tree of S to find all the symbols of $S[i_1, i_2]$ ($1 \le i_1 \le i_2 \le n$) which are in the range $[j_1, j_2]$ ($1 \le j_1 \le j_2 \le \sigma$). The operation takes $O(\log \sigma)$ to count the number of results [25], see Algorithm 1. This is easily modified to report each such occurrence in $O(\log \sigma)$ time by tracking each result upwards in the wavelet tree to find its position in S, and downwards to find its symbol in $[1, \sigma]$, just as in the original algorithm [25].

Algorithm: RANGE$(v, [i_1, i_2], [j_1, j_2], [t_1, t_2])$
if $i_1 > i_2$ **or** $[t_1, t_2] \cap [j_1, j_2] = \emptyset$ **then return** 0
if $[t_1, t_2] \subseteq [j_1, j_2]$ **then return** $i_2 - i_1 + 1$
$tm \leftarrow \lfloor (t_1 + t_2)/2 \rfloor$
$[i_{l1}, i_{l2}] \leftarrow [rank_{B_v}(0, i_1 - 1) + 1, rank_{B_v}(0, i_2)]$
$[i_{r1}, i_{r2}] \leftarrow [rank_{B_v}(1, i_1 - 1) + 1, rank_{B_v}(1, i_2)]$ $// = [i_1 - i_{l1}, i_2 - i_{l2}]$
return RANGE$(v_l, [i_{l1}, i_{l2}], [j_1, j_2], [t_1, t_m])$ + RANGE$(v_r, [i_{r1}, i_{r2}], [j_1, j_2], [t_m + 1, t_2])$

Algorithm 1: Range query algorithm: v is a wavelet tree node, B_v the bitmap stored at v, and v_l/v_r its left/right children. It is invoked with RANGE$(root, [i_1, i_2], [j_1, j_2], [1, \sigma])$.

2.3. Straight-Line Programs

We now define a Straight-Line Program (SLP) and highlight some properties.

Definition 2.1. [20] A *Straight-Line Program (SLP)* $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma)$ is a grammar that defines a single finite sequence $T[1, u]$, drawn from an alphabet $\Sigma = [1, \sigma]$ of terminals. It has n rules, which must be of the following types:

- $X_i \to \alpha$, where $\alpha \in \Sigma$. It represents string $\mathcal{F}(X_i) = \alpha$.
- $X_i \to X_l X_r$, where $l, r < i$. It represents string $\mathcal{F}(X_i) = \mathcal{F}(X_l)\mathcal{F}(X_r)$.

We call $\mathcal{F}(X_i)$ the *phrase generated* by nonterminal X_i, and $T = \mathcal{F}(X_n)$.

Definition 2.2. [37] The *height* of a symbol X_i in the SLP $\mathcal{G} = (X, \Sigma)$ is defined as $height(X_i) = 1$ if $X_i \to \alpha \in \Sigma$, and $height(X_i) = 1 + \max(height(X_l), height(X_r))$ if $X_i \to X_l X_r$. The height of the SLP is $height(\mathcal{G}) = height(X_n)$. We will refer to $height(\mathcal{G})$ as h when the referred grammar is clear from the context.

As some of our results will depend on the height of the SLP, it is interesting to recall the following theorem, which establishes the cost of balancing an SLP.

Theorem 2.1. [37] Let an SLP $\mathcal{G}$ generate text $T[1, u]$ with n rules. We can build an SLP $\mathcal{G}'$ generating T, of $O(n \log u)$ rules, with $height(\mathcal{G}') = O(\log u)$, in $O(n \log u)$ time.

Finally, as several grammar-compression methods are far from optimal [7], it is interesting that one can find in linear time a reasonable (and balanced) approximation.

Theorem 2.2. [37] Let $\mathcal{G}$ be the minimal SLP generating a text $T[1, u]$ over integer alphabet, with n rules. We can construct an SLP $\mathcal{G}'$ generating T, of $O(n \log u)$ rules, for which $height(\mathcal{G}') = O(\log u)$, in $O(u)$ time.

3. Labeled Binary Relations with Range Queries

In this section we introduce a data structure for labeled binary relations with range query capabilities. Consider a binary relation $\mathcal{R} \subseteq A \times B$, where $A = \{1, 2, \ldots, n_1\}$, $B = \{1, 2, \ldots, n_2\}$, a function $\mathcal{L} : A \times B \to L \cup \{\perp\}$, which maps every pair in $\mathcal{R}$ to a label in $L = \{1, 2, \ldots, \ell\}, \ell \geq 1$, and pairs not in $\mathcal{R}$ to $\perp$. We support the following queries:

- $\mathcal{L}(a, b)$.
- $A(b) = \{a,\ (a, b) \in \mathcal{R}\}$.
- $B(a) = \{b,\ (a, b) \in \mathcal{R}\}$.
- $R(a_1, a_2, b_1, b_2) = \{(a, b) \in \mathcal{R},\ a_1 \leq a \leq a_2,\ b_1 \leq b \leq b_2\}$.
- $\mathcal{L}(l) = \{(a, b) \in \mathcal{R},\ \mathcal{L}(a, b) = l\}$.
- The sizes of the sets: $|A(b)|$, $|B(a)|$, $|R(a_1, a_2, b_1, b_2)|$, and $|\mathcal{L}(l)|$.

A \ B	1	2	3
1	*1*	*2*	
2		*2*	*2*
3		*1*	

S_B	1	2		2	3		2	
$S_{\mathcal{L}}$	*1*	*2*		*2*	*2*		*1*	
X_B	0	0	1	0	0	1	0	1
X_A	0	1	0	0	0	1	0	1

Figure 1. Example of a labeled relation (left) and our representation of it (right). Labels are slanted and the elements of B are in typewriter font.

We build on an idea by Barbay et al. [4]. We define, for $a \in A$, $s(a) = b_1 b_2 \dots b_k$, where $b_i < b_{i+1}$ for $1 \leq i < k$ and $B(a) = \{b_1, b_2, \dots, b_k\}$. We build a string $S_B = s(1)s(2)\dots s(n_1)$ and write down the cardinality of each $B(a)$ in unary on a bitmap $X_B = 0^{|B(1)|}10^{|B(2)|}1 \dots 0^{|B(n_1)|}1$. We also store a bitmap $X_A = 0^{|A(1)|}10^{|A(2)|}1 \dots 0^{|A(n_2)|}1$. Another sequence $S_{\mathcal{L}}$ lists the labels $\mathcal{L}(a,b)$ in the same order they appear in S_B: $S_{\mathcal{L}} = l(1)l(2)\dots l(n_1)$, $l(a) = \mathcal{L}(a,b_1)\mathcal{L}(a,b_2)\dots\mathcal{L}(a,b_k)$. Figure 1 shows an example.

We represent S_B using wavelet trees [15], $\mathcal{L}$ with the structure for large alphabets [14], and X_A and X_B in compressed form [34]. Calling $r = |\mathcal{R}|$, S_B requires $r \log n_2 + o(r) \log n_2$ bits, $\mathcal{L}$ requires $r \log \ell + r\, o(\log \ell)$ bits (i.e., zero if $\ell = 1$), and X_A and X_B use $O(n_1 \log \frac{r+n_1}{n_1} + n_2 \log \frac{r+n_2}{n_2}) + o(r + n_1 + n_2) = O(r) + o(n_1 + n_2)$ bits.

We answer queries as follows. Let us define $map(a) = select_{X_B}(1, a-1) - (a-1)$ the function that gives the position in S_B preceding the area listing the elements of B associated to $a \in A$. Similarly, $unmap(p) = 1 + select_{X_B}(0,p) - p$ gives the row $a \in A$ associated to a position p of S_B. Both can be computed in constant time.

- $|A(b)|$: This is $select_{X_A}(1,b) - select_{X_A}(1,b-1) - 1$, the length of the area in X_A related to b.
- $|B(a)|$: It is computed in the same way using X_B. Note the formula is actually $map(b+1) - map(b)$.
- $\mathcal{L}(a,b)$: If $rank_{S_B}(b, map(a)) = rank_{S_B}(b, map(a+1))$ then a and b are not related and we return $\perp$, as $S_B[map(a)+1, map(a+1)]$ lists the elements related to a and b is not mentioned. Otherwise we return $S_{\mathcal{L}}[select_{S_B}(b, rank_{S_B}(b, map(a)) + 1)]$.
- $A(b)$: We first compute $|A(b)|$ and then retrieve the i-th element with $unmap(select_{S_B}(b,i))$, which gives the row a where each occurrence of b is mentioned in S_B, for $1 \leq i \leq |A(b)|$.
- $B(a)$: This is simply $S_B[map(a)+1, map(a+1)]$.
- $\mathcal{R}(a_1, a_2, b_1, b_2)$: We first determine which elements in S_B correspond to the range $[a_1, a_2]$: $[a_1', a_2'] = [map(a_1)+1, map(a_2)]$. Then, using the range query described in Section 2.2 on the wavelet tree of S_B, we count or retrieve the elements from $S_B[a_1', a_2']$ which are in the range $[b_1, b_2]$.
- $\mathcal{L}(l)$: We retrieve consecutive occurrences $y_i = select_{S_{\mathcal{L}}}(l,i)$ of l in $S_{\mathcal{L}}$, reporting the corresponding pairs $(a,b) = (unmap(y_i), S_B[y_i])$. Determining $|\mathcal{L}(l)|$ is done via $rank_{S_{\mathcal{L}}}(l,r)$.

We note that, if we do not support queries $\mathcal{R}(a_1, a_2, b_1, b_2)$, we can use also the faster data structure [14] for S_B. We have thus proved the next theorem.

Theorem 3.1. Let $\mathcal{R} \subseteq A \times B$ be a binary relation, where $A = \{1, 2, \ldots, n_1\}$, $B = \{1, 2, \ldots, n_2\}$, and a function $\mathcal{L} : A \times B \to L \cup \{\perp\}$, which maps every pair in $\mathcal{R}$ to a label in $L = \{1, 2, \ldots, \ell\}$, $\ell \geq 1$, and pairs not in $\mathcal{R}$ to $\perp$. Then $\mathcal{R}$ can be indexed using $(r+o(r))(\log n_2+\log \ell+o(\log \ell)+O(1))+o(n_1+n_2)$ bits of space, where $r = |\mathcal{R}|$. Queries can be answered in the times shown below, where k is the size of the output. One can choose (i) $rnk(x) = acc(x) = \log\log x$ and $sel(x) = 1$, or (ii) $rnk(x) = \log\log x \log\log\log x$, $acc(x) = 1$ and $sel(x) = \log\log x$, independently for $x = \ell$ and for $x = n_2$.

Operation	Time (with range)	Time (without range)
$\mathcal{L}(a, b)$	$O(\log n_2 + acc(\ell))$	$O(rnk(n_2) + sel(n_2) + acc(\ell))$
$A(b)$	$O(1 + k \log n_2)$	$O(1 + k\, sel(n_2))$
$B(a)$	$O(1 + k \log n_2)$	$O(1 + k\, acc(n_2))$
$\lvert A(b)\rvert, \lvert B(a)\rvert$	$O(1)$	$O(1)$
$R(a_1, a_2, b_1, b_2)$	$O((k + 1) \log n_2)$	—
$\lvert R(a_1, a_2, b_1, b_2)\rvert$	$O(\log n_2)$	—
$\mathcal{L}(l)$	$O((k + 1)sel(\ell) + k \log n_2)$	$O((k + 1)sel(\ell) + k\, acc(n_2))$
$\lvert\mathcal{L}(l)\rvert$	$O(rnk(\ell))$	$O(rnk(\ell))$

We note the asymmetry of the space and time with respect to n_1 and n_2, whereas the functionality is symmetric. This makes it always convenient to arrange that $n_1 \geq n_2$.

4. A Powerful SLP Representation

We provide in this section an SLP representation that supports various queries on the SLP within essentially the same space of a plain representation.

Recalling that Σ is the alphabet of the SLP and σ its size, it will usually be the case that all the symbols of Σ are used in the SLP. Otherwise, we can use a bitmap $C[1, \sigma]$ marking the symbols of Σ that are used in the SLP. We can use $select_C(1, i)$ to find the i-th alphabet symbol used in the SLP and $rank_C(1, x)$ to find the rank of symbol x in a contiguous list of those used in the SLP. By using Raman et al.'s representation [34], C requires at most $n \log \frac{\sigma}{n} + O(n) + o(\sigma)$ bits, while supporting $rank$ and $select$ on C in constant time. Thus we can assume that the alphabet used is contiguous in $[1, \sigma]$.

We will assume that the rules of the form $X_i \to \alpha$ are lexicographically sorted, that is, if there is another rule $X_j \to \beta$, then $i < j$ if and only if $\alpha < \beta$. The SLP can obviously be reordered so that this holds. If for some reason we need to retain the original one, $\sigma \log \sigma$ extra bits are needed to record the rule reordering.

A plain representation of an SLP with n rules over effective alphabet $[1, \sigma]$ requires at least $2(n - \sigma)\lceil \log n\rceil + \sigma\lceil \log \sigma\rceil \leq 2n\lceil \log n\rceil$ bits. Based on our labeled binary relation data structure of Theorem 3.1, we give now an alternative SLP representation which requires asymptotically the same space, $2n \log n + o(n \log n)$ bits, and is able to answer a number of interesting queries on the grammar in $O(\log n)$ time. This will be a key part of our indexed SLP representation.

We regard again a binary relation as a table where the rows represent the elements of set A and the columns the elements of B. In our representation, every row corresponds to a symbol X_l (set A) and every column to a symbol X_r (set B). Pairs (l, r) are related, with label i, whenever there exists a rule $X_i \to X_l X_r$. Since $A = B = L = \{1, 2, \ldots n\}$ and $|\mathcal{R}| = n$, the structure uses $2n \log n + o(n \log n)$ bits. We note also that function $\mathcal{L}$ is invertible, thus $|\mathcal{L}(l)| = 1$.

To handle the rules of the form $X_i \to \alpha$, we set up a bitmap $Y[1, n]$ so that $Y[i] = 1$ if and only if $X_i \to \alpha$ for some $\alpha \in \Sigma$. Thus we know $X_i \to \alpha$ in constant time because $Y[i] = 1$ and $\alpha = rank_Y(1, i)$. The total space is $n + o(n) = O(n)$ bits [8]. This works because these rules are lexicographically sorted and all the symbols in Σ are used; we have already explained how to proceed otherwise.

This representation supports the following queries.

- *Access to rules:* Given i, find l and r such that $X_i \to X_l X_r$, or α such that $X_i \to \alpha$. If $Y[i] = 1$ we obtain α in constant time as explained. Otherwise, we obtain $\mathcal{L}(i) = \{(l, r)\}$ from the labeled binary relation, in $O(\log n)$ time.

- *Reverse access to rules:* Given l and r, find i such that $X_i \to X_l X_r$, if any. This is done in $O(\log n)$ time via $\mathcal{L}(l, r)$ (if it returns $\perp$, there is no such X_i). We can also find, given α, the $X_i \to \alpha$, if any, in $O(1)$ time via $i = select_Y(1, \alpha)$.

- *Rules using a left/right symbol:* Given i, find those j such that $X_j \to X_i X_r$ (left) or $X_j \to X_l X_i$ (right) for some X_l, X_r. The first is answered using $\{\mathcal{L}(i, r), r \in B(i)\}$ and the second using $\{\mathcal{L}(l, i), l \in A(i)\}$, in $O(\log n)$ time per each j found.

- *Rules using a range of symbols:* Given $l_1 \le l_2, r_1 \le r_2$, find those i such that $X_i \to X_l X_r$ for any $l_1 \le l \le l_2$ and $r_1 \le r \le r_2$. This is answered, in $O(\log n)$ time per symbol retrieved, using $\{\mathcal{L}(a, b), (a, b) \in \mathcal{R}(l_1, l_2, r_1, r_2)\}$.

Again, if the last operation is not provided, we can choose the faster representation [14] (alternative (i) in Theorem 3.1), to achieve $O(\log \log n)$ time for all the other queries. Or, if we want to provide "access to rules" in constant time as a plain SLP representation, we choose (i) for $S_{\mathcal{L}}$ and (ii) for S_B, obtaining $O(\log \log n \log \log \log n)$ time for the other operations.

Theorem 4.1. An SLP $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma)$, $\Sigma = [1, \sigma]$, can be represented using $2n \log n + o(\sigma + n \log n)$ bits, such that all the queries described above (access to rules, reverse access to rules, rules using a symbol, and rules using a range of symbols) can be answered in $O(\log n)$ time per delivered datum. If we do not support the rules using a range of symbols, times drop to $O(\log \log n)$, or to $O(1)$ for access to rules and $O(\log \log n \log \log \log n)$ for the others.

5. Indexable Grammar Representations

We now provide an SLP-based text representation that permits indexed search and random access. We assume our text $T[1, u]$, over alphabet $\Sigma = [1, \sigma]$, is represented with an SLP $\mathcal{G}$of n rules.

We will represent $\mathcal{G}$ using a variant of Theorem 4.1, where we apply some reorderings to the rules. First, we will reorder all the rules in lexicographic order of the strings represented, that is,

$\mathcal{F}(X_i) \leq \mathcal{F}(X_{i+1})$ for all $1 \leq i < n$. Therefore the columns of the binary relation will represent X_r, yet lexicographically sorted by $\mathcal{F}(X_r)$. Instead, the rows will represent X_l sorted by *reverse* lexicographic order, that is lexicographically sorted by $\mathcal{F}(X_l)^{rev}$, where S^{rev} is string S read backwards. We will also store a permutation π, which maps reverse to direct lexicographic ordering. This must be used to translate row positions to nonterminal identifiers (as these are sorted in direct lexicographical order). We use Munro et al.'s representation [30] for π, with parameter $\epsilon = \frac{1}{\log n}$, so that π can be computed in constant time and π^{-1} in $O(\log n)$ time, and the structure needs $n \log n + O(n)$ bits of space.

With the SLP representation and π, the space required is $3n \log n + o(\sigma + n \log n)$ bits. We add other $n \log u$ bits for storing the lengths $|\mathcal{F}(X_i)|$ of all the nonterminals X_i. Note that our reordering preserves the lexicographic ordering of the rules $X_i \rightarrow \alpha$, needed for our binary relation based representation.

Figure 2 gives an example grammar representation. Disregard for now the arrows and shadings, which illustrate the search process.

5.1. Extraction of Text from an SLP

To expand a substring $\mathcal{F}(X_i)[j, j']$, we first find position j: We recursively descend in the parse tree rooted at X_i until finding its jth position. Let $X_i \rightarrow X_l X_r$, then if $|\mathcal{F}(X_l)| \geq j$ we descend to X_l, otherwise to X_r, in this case looking for position $j - |\mathcal{F}(X_l)|$. This takes $O(height(X_i) \log n)$ time (where the $\log n$ factor is the time for "access to rules" operation). In our way back from the recursion, if we return from the left child, we fully traverse the right child left to right, until outputting $j' - j + 1$ terminals.

This takes in total $O((height(X_i) + j' - j) \log n)$ time, which is at most $O((h + j' - j) \log n)$. This is because, on one hand, we will follow both children of a rule at most $j' - j$ times. On the other, we will follow only one child at most twice per tree level, as otherwise two of them would share the same parent.

5.2. Searching for a Pattern in an SLP

The problem is to find all the occurrences of a pattern $P = p_1 p_2 \ldots p_m$ in the text $T[1, u]$ defined by an SLP of n rules. As in previous work [19], except for the special case $m = 1$, occurrences can be divided into *primary* and *secondary*. A primary occurrence in $\mathcal{F}(X_i)$, $X_i \rightarrow X_l X_r$, is such that it spans a suffix of $\mathcal{F}(X_l)$ and a prefix of $\mathcal{F}(X_r)$, whereas each time X_i is used elsewhere (directly or transitively in other nonterminals that include it) it produces secondary occurrences. In the case $P = \alpha$, we say that the only primary occurrence is at $X_i \rightarrow \alpha$ and the other occurrences are secondary.

Our strategy is to first locate the primary occurrences, and then track all their secondary occurrences in a recursive fashion. To find primary occurrences of P, we test each of the $m - 1$ possible partitions $P = P_l P_r$, $P_l = p_1 p_2 \ldots p_k$ and $P_r = p_{k+1} \ldots p_m$, $1 \leq k < m$. For each partition $P_l P_r$, we first find all those X_ls such that P_l is a suffix of $\mathcal{F}(X_l)$, and all those X_rs such that P_r is a prefix of $\mathcal{F}(X_r)$. The latter forms a lexicographic range $[r_1, r_2]$ in the $\mathcal{F}(X_r)$s, and the former a lexicographic range $[l_1, l_2]$ in the $\mathcal{F}(X_l)^{rev}$s. Thus, using our SLP representation, the X_is containing the primary occurrences correspond those labels i found within rows l_1 and l_2, and between columns r_1 and r_2, of the binary relation. Hence a query for *rules using a range of symbols* will retrieve each such X_i in $O(\log n)$ time. If $P = \alpha$, our only primary occurrence is obtained in $O(1)$ time using *reverse access to rules*.

$T = abracadabra$

	$\mathcal{F}$
$A \to a$	a
$B \to b$	b
$C \to c$	c
$D \to d$	d
$R \to r$	r
$U \to AB$	ab
$V \to RA$	ra
$W \to UV$	$abra$
$X \to CA$	ca
$Y \to DW$	$dabra$
$Z \to WX$	$abraca$
$S \to ZY$	$abracadabra$

$\Longrightarrow$ SORT

initial symbol: X_5

	$\mathcal{F}$
$X_1 \to a$	a
$X_2 \to X_1X_6$	ab
$X_3 \to X_2X_{12}$	$abra$
$X_4 \to X_3X_8$	$abraca$
$X_5 \to X_4X_{10}$	$abracadabra$
$X_6 \to b$	b
$X_7 \to c$	c
$X_8 \to X_7X_1$	ca
$X_9 \to d$	d
$X_{10} \to X_9X_3$	$dabra$
$X_{11} \to r$	r
$X_{12} \to X_{11}X_1$	ra

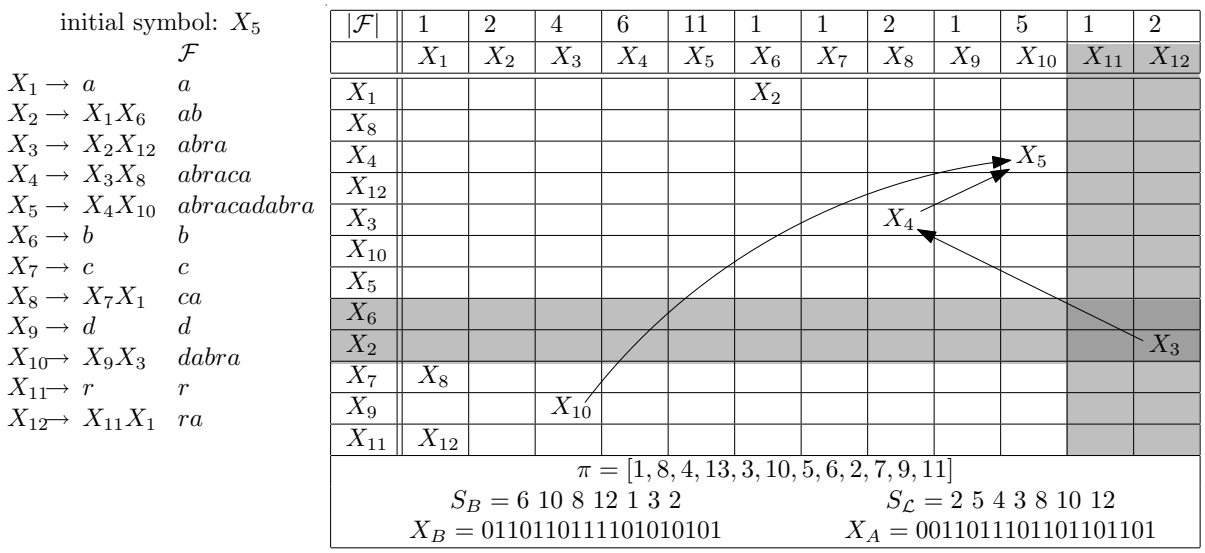

Figure 2. An example grammar for the text T = `"abracadabra"`. On top, the nonterminals are renamed according to their lexicographic order, so that A corresponds to X_1, U to X_2, and so on. On the bottom, our data structure representing T. What the index stores is S_B, $S_{\mathcal{L}}$, X_B, X_A, π, and $|\mathcal{F}|$; all the rest is given for illustrative purposes (we omit bitmap Y). We also illustrate the search process for P = `"br"`: We search the rows for the nonterminals finishing with `"b"` and the columns for the nonterminals starting with `"r"`. The intersection contains X_3 (formerly W), where P has its only primary occurrence. The arrows show how we look for the rows and columns corresponding to X_3, to find out that it is used within X_4 (formerly Z) and X_{10} (formerly Y), and these in turn yield the two occurrences within X_5, the initial symbol.

Now, given each primary occurrence at X_i, we must track all the nonterminals that use X_i in their right hand sides. As we track the occurrences, we also maintain the *offset* of the occurrence within the nonterminal. The offset for the primary occurrence at $X_i \to X_l X_r$ is $|\mathcal{F}(X_l)| - k + 1$ (l is obtained with an *access to rule* query for i). Each time we arrive at the initial symbol X_s, the offset gives the position of a new occurrence.

To track the uses of X_i, we first find all those $X_j \to X_i X_r$ for some X_r, using query *rules using a left symbol* for $\pi^{-1}(i)$. The offset is unaltered within those new nonterminals. Second, we find all those $X_j \to X_l X_i$ for some X_l, using query *rules using a right symbol* for i. The offset in these new nonterminals is that within X_i plus $|\mathcal{F}(X_l)|$, where again $\pi^{-1}(l)$ is obtained from the result using an *access to rule* query, and then we apply π to get l. We proceed recursively with all the nonterminals X_j found, reporting the offsets (and finishing) each time we arrive at X_s.

Note that we are tracking each occurrence individually, so that we can process several times the same nonterminal X_i, yet with different offsets. Each occurrence may require to traverse all the syntax tree up to the root, and we spend $O(\log n)$ time at each step. Moreover, we carry out $m-1$ range queries for the different pattern partitions. Thus the overall time to find the occ occurrences is $O((m + h\,occ)\log n)$.

We remark that we do not need to output all the occurrences of P. If we just want occ occurrences, our cost is proportional to this occ. Moreover, the *existence problem*, that is, determining whether or not P occurs in T, can be answered just by considering whether or not there are any primary occurrences.

Figure 2 illustrates the search process. There is a remaining problem, however: How to find the range of phrases starting/ending with a suffix/prefix of P. This is considered next.

5.3. Prefix and Suffix Searching

We present different time/space tradeoffs to search for P_l and P_r in the respective sets.

Binary search based approach. We can perform a binary search over the $\mathcal{F}(X_i)$s and over the $\mathcal{F}(X_i)^{rev}$s to determine the ranges where P_r and P_l^{rev}, respectively, belong. In order to do the string comparisons in the first binary search, we extract the first at most m terminals of $\mathcal{F}(X_i)$, in time $O((m + h)\log n)$ (Section 5.1). As the binary search requires $O(\log n)$ comparisons, the total cost is $O((m + h)\log^2 n)$ for the partition $P_l P_r$. The search within the reverse phrases is similar, except that we extract the at most m rightmost terminals and must use π to find the rule from the position in the reverse ordering. This variant needs no extra space.

Compact Patricia Trees. Another option is to build Patricia Trees [29] for the $\mathcal{F}(X_i)$s and for the $\mathcal{F}(X_i)^{rev}$s (adding them a terminator so that each phrase corresponds to a leaf). A Patricia tree is a binary digital tree where each root-to-leaf path spells out one string (where the character values are converted to binary), and unary paths are removed by storing at each node the number of bits skipped from its parent. A search can proceed normally on the explicit bits, and a final check against any leaf of the subtree found is needed to verify the matching of the skipped bits in the search path.

By using the cardinal tree representation of Benoit et al. [6] for the tree structure and the edge labels, each such tree can be represented using $2n \log \sigma + O(n)$ bits, and traversal (including to a child labeled α) can be carried out in constant time. The ith leaf of the tree for the $\mathcal{F}(X_i)$s corresponds to nonterminal X_i (and the ith of the tree for the $\mathcal{F}(X_i)^{rev}$s, to $X_{\pi(i)}$). Hence, upon reaching the tree node corresponding

to the search string, we obtain the lexicographic range by counting the number of leaves up to the node subtree and past it, which can also be done in constant time [6].

The difficult point is how to store the Patricia tree skips, as in principle they require other $4n \log u$ bits of space. If we do not store the skips at all, we can still compute them at each node by extracting the corresponding substrings for the leftmost and rightmost descendant of the node, and checking for how many more symbols they coincide [8]. This can be obtained in time $O((\ell + h) \log n)$, where ℓ is the skip value (Section 5.1). The total search time is thus $O(m \log n + mh \log n) = O(mh \log n)$.

Instead, we can use k bits for the skips, so that skips in $[1, 2^k - 1]$ can be represented, and a skip zero means $\geq 2^k$. Now we need to extract leftmost and rightmost descendants only when the edge length is $\ell \geq 2^k$, and we will work $O((\ell - 2^k + h) \log n)$ time. Although the $\ell - 2^k$ terms still can add up to $O(m)$ (e.g., if all the lengths are $\ell = 2^{k+1}$), the h terms can be paid only $O(1 + m/2^k)$ times. Hence the total search cost is $O((m + h + \frac{mh}{2^k}) \log n)$, at the price of at most $4nk$ extra bits of space. We must also do the final Patricia tree check due to skipped characters, but this adds only $O((m + h) \log n)$ time. For example, using $k = \log h$ we get $O((m + h) \log n)$ time and $4n \log h$ extra bits of space.

As we carry out $m - 1$ searches for prefixes and suffixes of P, as well as $m - 1$ range searches, plus *occ* extraction of occurrences, we have the final result.

Theorem 5.1. Let $T[1, u]$ be a text over alphabet $[1, \sigma]$ represented by an SLP of n rules and height h. Then there exists a representation of T using $n(\log u + 3 \log n + O(\log \sigma + \log h) + o(\log n)) + o(\sigma)$ bits, such that any substring $T[l, r]$ can be extracted in time $O((r - l + h) \log n)$, and the positions of the occurrences of a pattern $P[1, m]$ in T can be located in a fixed time $O(m(m + h) \log n)$ plus $O(h \log n)$ time per occurrence reported. By removing the $O(\log h)$ term in the space, the fixed locating time raises to $O(m^2 h \log n)$. By further removing the $O(\log \sigma)$ term in the space, that time raises to $O(m(m + h) \log^2 n)$. The existence problem is solved within the fixed locating time.

Compared with the $2n \log n$ bits of the plain SLP representation, ours requires at least $4n \log n + o(n \log n)$ bits, that is, roughly twice the space. More generally, as long as $u = n^{O(1)}$, our representation uses $O(n \log n)$ bits, of the same order of the SLP size. Otherwise, our representation is superlinear in the size of the SLP (almost quadratic in the extreme case $n = O(\log u)$). Yet, if $u = n^{\omega(1)}$, our representation takes $u^{o(1)}$ bits, which is still much smaller than the *original* text.

Combining both methods. We can combine the two previous approaches as follows. We build the top part of the Patricia trees, until the subtree size is between k and $2k$, for some parameter k. At that point we switch to binary search on the remaining range. For the part of the Patricia tree we do store the skips, and we also store the leaf ranges corresponding to the subtrees. Now the search can be done with the usual Patricia tree algorithm (up to a certain point in the string) in time $O(m)$ for descending plus $O((m + h) \log n)$ for checking the string (due to the skipped characters). This search yields a range of Patricia tree leaves, which must be completed with two binary searches over $O(k)$ strings, one per extreme of the leaf range, to obtain the precise range of strings. These binary searches require time $O((m + h) \log n \log k)$, which dominates the overall complexity. In exchange, the space is $O(\frac{n}{k} \log u)$. This leads to the following theorem.

Theorem 5.2. Let $T[1, u]$ be a text over alphabet $[1, \sigma]$ represented by an SLP of n rules and height h. Then there exists a representation of T using $n(\log u + 3 \log n + O(\log(u)/k) + o(\log n)) + o(\sigma)$ bits,

k	time	space (bits)
$\Theta(\log u)$	$O(m(m+h)\log n \log\log u)$	$O(n)$
$\Theta(\log\log u)$	$O(m(m+h)\log n \log\log\log u)$	$n\, o(\log u)$
$\Theta(\frac{\log u}{\log n})$	$O(m(m+h)\log u)$	$O(n\log n)$
$\Theta(\log n)$	$O(m(m+h)\log^2 n)$	$o(n)\ \log u$
$\Theta(1)$	$O(m(m+h)\log n)$	$O(n\log u)$

Table 1. Different tradeoffs for the combination of compact patricia trees and the binary search based approach.

for any parameter $1 \leq k \leq \log u$, such that any substring $T[l,r]$ can be extracted in time $O((r-l+h)\log n)$, and the positions of the occurrences of a pattern $P[1,m]$ in T can be located in a fixed time $O(m(m+h)\log n \log k)$ plus $O(h\log n)$ time per occurrence reported. The existence problem is solved within the fixed locating time.

Some examples of the trade-off offered by this solution are shown in Table 1.

5.4. Construction

We discuss now how to carry out the construction of our index given the SLP.

For the binary relation that represents the grammar, assuming we already have the nonterminals X_l and X_r in the proper order, we first create one list per X_l, and then traverse the rules $X_i \to X_l X_r$ in X_r order, adding pair (X_r, X_i) to the end of list X_l. Then we traverse the lists in X_l order, adding the X_r components to S_B and the X_is to $S_{\mathcal{L}}$. All this takes $O(n\log n)$ time, dominated by the ordering the rules in X_r order. Bitvectors X_A and X_B are easily built in $O(n)$ time, including their $rank$/$select$ structures.

Building the wavelet trees for S_B and $S_{\mathcal{L}}$ takes $O(n\log n)$ additional time. The wavelet trees are built in linear time per level, and the reordered children for the next level are also obtained in linear time using the bits of the bitvector of the current level. There are $O(\log n)$ levels, which leads to the $O(n\log n)$ construction time.

The lengths $|\mathcal{F}(X_i)|$ are easily obtained in $O(n)$ time, by performing a bottom-up traversal of the DAG of the grammar (going first top-down, marking the already traversed nodes to avoid retraversing, and assigning the lengths in the return of the recursion).

The remaining cost is that of lexicographically sorting the strings $\mathcal{F}(X_r)$ and $\mathcal{F}(X_l)^{rev}$, or alternatively, building the tries. In principle this can take as much as $\sum_{i=1}^{n} |\mathcal{F}(X_i)|$, which can be even $\omega(u)$. Let us focus on sorting the direct phrases $\mathcal{F}(X_i)$, as the reversed ones can be handled identically.

Our solution is based on the fact that all the phrases are substrings of $T[1,u]$. We first build the *suffix array* [28] of T in $O(u)$ time [18]. This is an array $A[1,u]$ pointing to all the suffixes of $T[1,u]$ in lexicographic ordering, $T[A[i],u] < T[A[i+1],u]$. As it is a permutation, we can also build its inverse $A^{-1}[1,u]$ in $O(u)$ time.

We now expand T using the grammar, so as to record one starting text position p_i for each string $\mathcal{F}(X_i)$, and then sort the X_is in $O(n\log n)$ time using A^{-1}: X_i is smaller than X_j iff $A^{-1}[p_i] < A^{-1}[p_j]$. This will correctly order all the $\mathcal{F}(X_i)$ strings unless one is a prefix of the other, but in this case the ordering is not relevant for our algorithm.

To build the Patricia trees, instead, we build the *suffix tree* of T in $O(u)$ time [9]. This can be seen as a Patricia tree built on all the suffixes of T. We mark the n suffix tree leaves corresponding to phrase beginnings (that is, the $A^{-1}[p_i]$-th leaves), and create new leaves at depth $|\mathcal{F}(X_i)|$ which are ancestors of the marked leaves. The point to insert these n new leaves are found by binary searching the string depths $|\mathcal{F}(X_i)|$ with level ancestor queries [5] from the marked suffix tree leaves. Finally, the desired Patricia tree is formed by collecting the ancestor nodes of the new leaves while collapsing unary paths again, which yields $O(n)$ nodes.

The whole process takes $O(u + n \log n)$ time and $O(u \log u)$ bits of space.

5.5. Faster Locating and Counting

We are right now locating each occurrence individually, even if they share the same phrase (albeit with different offsets). We show now that, if one can have some extra space for the query process, the $O(h\, occ \log n)$ time needed for the occ occurrences can be turned to $O(\min(h\, occ, n) \log n + occ)$, thus reducing the time when there are many occurrences to report.

We set up a min-priority queue H, where we insert the phrases X_i where primary occurrences are found. We do not yet propagate those to secondary occurrences. The priority of X_i will be $|\mathcal{F}(X_i)|$. For each such X_i, with $X_i \to X_l X_r$, we store l and r; the minimum and maximum offset of the primary occurrences found in X_i; left and right *pointers*, initially null and later pointing to the data for X_l and X_r, if they are eventually inserted in H; and left and right offsets associated to those pointers. The data of those X_i will be kept in a fixed memory position across all the process, so we can set pointers to them, which will be valid even after we remove them from H (H contains just pointers to those memory areas). The left and right pointers point to those areas as well. Separately, we store a balanced binary search tree that, given i, gives the memory position of X_i, if it exists (this tree permits, in particular, freeing all the memory areas at the end).

Now, we repeatedly extract an element X_i with smallest $|\mathcal{F}(X_i)|$ from H, and find using our binary relation data structures all the other X_js that mention X_i in their rule. We use the balanced tree to determine whether X_j is already in H (and where is its memory area) or not. In the second case, we allocate memory for X_j and insert it into H (note that X_j could already be in H, for example if it has its own primary occurrences). Now, if $X_j \to X_i X_r$, then we set the left pointer of X_j (1) to the left pointer of X_i if X_i does not have primary occurrences nor right pointer, setting the left offset of X_j to that of X_i; (2) to the right pointer of X_i if X_i does not have primary occurrences nor left pointer, setting the left offset of X_j to the right offset of X_i; (3) to X_i itself otherwise, setting the left offset of X_i to zero. If $X_j \to X_l X_i$, we assign the right pointer and offset of X_j in the same way, except that we add $|\mathcal{F}(X_l)|$ to the right offset of X_j. Note that the priority queue ordering implies that all the occurrences descending from X_j are already processed when we process X_j itself.

The process finishes when we extract the initial symbol from H and H becomes empty. At this point we are ready to report all of the occurrences with a recursive procedure starting at the initial symbol. Moreover, we can report them in text order: To report $X_i \to X_l X_r$, we first report the occurrences at the left pointer of X_i (if not null), shifting their values by the left offset of X_i; then the primary occurrences of X_i (if any); and then the occurrences at the right pointer of X_i (if not null), shifting their values by the right offset of X_i. Those shifts accumulate as recursion goes down the tree, and become the true occurrence positions at the end.

To display all the primary occurrences of a node knowing only the *first* and *last* positions, we notice

that these occurrences must overlap, thus we know the full text content of the area where the primary occurrences other than the first and the last may appear. By preprocessing the pattern we can obtain those occurrences in constant time each: Let $last - first = d$, so $last - d$ is the *first* primary occurrence. This means that $P[d+1, m] = P[1, m-d]$, thus P occurs at positions 1 (*first*) and $d+1$ (*last*) of string $X = P[1,d] \cdot P = P \cdot P[m-d+1, m]$. We wish to know which is the occurrence of P in X that precedes that at position $d+1$. We can search for P in $X[1, m+d-1]$ in time $O(m)$ using algorithm KMP [23] and store the position $d' < d$ of the last occurrence in a table $O[d]$. For the second previous occurrence, we have already that P occurs at position $d'+1$ of string $X' = P[1,d'] \cdot P$, thus it corresponds to $O[d']$.

Therefore, it is enough to precompute all those $O[1,m]$ values in $O(m^2)$ time beforehand. Later, given *first* and *last*, we report each primary occurrence in constant time by doing $d \leftarrow last - first$, reporting $last - d$, then $d \leftarrow O[d]$, reporting $last - d$, and so on until $d = 0$, where we report $last$.

Let us now analyze the algorithm. Although each occurrence can trigger h insertions into H, nodes are not repeated in H, and thus there are at most $O(\min(h\,occ, n))$ elements in H. Thus the space is in the worst case $O(\min(h\,occ, n) \log u + m \log m)$ bits (the second part is for O). As for the time, we pay $O(\log n)$ time to insert each primary occurrence into H and compute its associated data, $O(\log n)$ time to extract it from H, and $O(\log n)$ time to find each of its parents and insert them into H (each parent $X_i \rightarrow X_l X_r$ is processed at most twice, from X_l and from X_r). Thus the overall cost of filling and emptying H is $O(\min(h\,occ, n) \log n)$.

As for the process of reporting once H is emptied, note that the left and right pointers can be traversed in constant time and, because in the tree induced by the left/right pointers each pointed node either has at least one distinct primary occurrence, or it has two children, it follows that the total traversal time is $O(occ)$. Reporting all the primary occurrences can also be done in time $O(occ)$.

Overall, the time is $O(m^2 + \min(h\,occ, n) \log n + occ)$, provided we can afford the extra space at search time. Note the $O(m^2)$ part (to build $O[1,m]$) is dominated by higher terms in the search complexity.

Theorem 5.3. Under the same conditions of Theorems 5.1 and 5.2, we can locate the occ occurrences of $P[1,m]$ within the fixed locating time reported in those theorems plus $O(\min(h\,occ, n) \log n + occ)$, by using $O(\min(h\,occ, n) \log u + m \log m)$ extra bits of space at search time.

A simplification of this technique lets us count the number of occurrences of $P[1,m]$ more efficiently than by locating them all. We follow the same process of detecting the primary occurrences and using a heap to process the nonterminals by increasing length. We store, for each nonterminal, the number of occurrences of P inside it (initially zero). Each primary occurrence adds 1 to the counter of the corresponding nonterminal. Each nonterminal we extract from the heap adds its counter value to that of all the nonterminals that use it. When we finally extract the initial symbol, its counter is the number of occurrences of P. It is easy to see that the overall additional counting time is $O(\min(h\,occ, n) \log n)$, which is interesting when dominated by $O(n \log n)$ (otherwise it is better to locate the occurrences one by one).

Theorem 5.4. Under the same conditions of Theorems 5.1 and 5.2, we can count the occurrences of $P[1,m]$ within the fixed locating time reported in those theorems plus $O(n \log n)$, by using $O(n \log u)$ extra bits of space at search time.

Incidentally, this result provides an improved solution to a recently proposed problem on SLPs [17].

Corollary 5.1. Given a text $T[1,u]$ over an alphabet of size σ, and an SLP of n rules generating T, the problem of finding the most repeated substring of T of length at least two can be solved using $O(n \log u)$ bits of space and $O(\sigma^2 n \log n)$ time.

Proof:
Clearly it is sufficient to try with the substrings of length 2, as longer ones cannot be more frequent. We simply count the occurrences of all the $O(\sigma^2)$ possible pairs of characters, applying Theorem 5.4 with $m = 2$ using our fastest SLP representation of Theorem 5.1. □

The best previous result [17] needs $O(\sigma^2 n^2)$ time and $O(n^2)$ words of space, thus we significantly improve it in both aspects.

6. More General Grammars

Until now we have considered the case where the grammar-based compressor generates a single nonterminal symbol that represents the text. Many grammar-based compressors [41, 24, 33] output instead a *set of rules* (which can be seen as a forest of parse trees) and a *sequence* of terminals and nonterminals, whose expansion using the rules leads to the original text. This is captured by the following definition.

Definition 6.1. (Relaxed Straight-Line Program (RSLP))
A *Relaxed Straight-Line Program (RSLP)* $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma, \mathcal{C} = C_1 C_2 \ldots C_c)$ is a tuple where $\mathcal{G}$ is a grammar on an alphabet $\Sigma = [1, \sigma]$ of terminals, such that each X_i generates a single finite string $\mathcal{F}(X_i)$, and can be of two types, as follows:

- $X_i \to \alpha$, where $\alpha \in \Sigma$. It generates string $\mathcal{F}(X_i) = \alpha$.
- $X_i \to X_l X_r$, where $l, r < i$. It generates string $\mathcal{F}(X_i) = \mathcal{F}(X_l)\mathcal{F}(X_r)$.

Moreover, $\mathcal{C}$ is a sequence of terminals and nonterminals $C_i \in X \cup \Sigma$, and $\mathcal{G}$ represents the text $T[1,u] = \mathcal{F}(C_1)\mathcal{F}(C_2)\ldots\mathcal{F}(C_c)$, assuming $\mathcal{F}(\alpha) = \alpha$ for $\alpha \in \Sigma$.

It is clear that an RSLP can be converted into an SLP by adding $c-1$ new rules that derive $\mathcal{C}$ from an initial symbol I, and then the symbols of $\mathcal{C}$ expand as usual. The new rules can be balanced, thus adding only $\log c$ to the height of the grammar. We might also need to introduce nonterminals for any terminal that could be mentioned in $\mathcal{C}$.

Definition 6.2. The height of RSLP $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma, \mathcal{C})$ is $height(\mathcal{G}) = \max\{height(X_i), 1 \le i \le n\}$. We will refer to $height(\mathcal{G})$ as h when the referred grammar is clear from the context.

In the following, let $n' \le n + \min(c, \sigma)$ the number of rules in X once we add those of the form $X_i \to \alpha$ for the terminals α mentioned directly in $\mathcal{C}$ and not in X.

Corollary 6.1. Let $T[1,u]$ be a text over alphabet $[1,\sigma]$ represented by an RSLP of n' rules and height h, and a sequence of c nonterminal symbols. Then it can be represented using Theorems 5.1 or 5.2, using an SLP of $n' + c - 1$ rules and height $h + \lceil \log c \rceil + 1$. For example, it can be represented using $(n'+c)(\log u + 3\log(n'+c) + o(\log(n'+c))) + o(\sigma)$ bits, such that any substring $T[l,r]$ can be extracted in time $O((r - l + h + \log c)\log(n'+c))$, and the positions of the occurrences of a pattern $P[1,m]$ in T

can be located in a fixed time $O(m(m+h+\log c)\log^2(n'+c))$ plus $O((h+\log c)\log(n'+c))$ time per occurrence reported.

We propose now a more sophisticated scheme that can achieve better results.

1. We use our binary relation data structure to represent the forest of rules X. Thus it will require $n'(\log u + 3\log n' + o(\log n')) + o(\sigma)$ bits of space, according to Section 5.

2. The sequence $\mathcal{C}$ is represented with the structure for sequences over large alphabets [14]. This will require $c(\log n' + o(\log n'))$ bits of space and carry out *access* in $O(\log\log n')$ time and *select* in $O(1)$ time.

3. We store a bitmap $B[1,u]$ marking the positions of T where the symbols of $\mathcal{C}$ begin. It can be represented in compressed form [16, Theorem 17 p. 153] such that it uses $c\log\frac{u}{c} + O(c\log\log\frac{u}{c})$ bits and supports *rank* and *select* in $O(\log c)$ time.

4. We store another labeled binary relation of n' rows and c columns. Value $1 \le i \le n'$ is related to $1 \le j \le c$ with label $2 \le k \le c$ if the suffix of T that starts at the k-th symbol of $\mathcal{C}$ is at lexicographical position j among all such suffixes, and the lexicographic position of $\mathcal{F}(C_{k-1})^{rev}$, among all the distinct reversed nonterminals (and terminals) $\mathcal{F}(X_i)^{rev}$, is i. We wish to carry out range searches on this binary relation. Yet, as there is exactly one point per column, we do not need the bitmap X_A, and moreover store $S_{\mathcal{L}}$ in column-wise order, rather than row-wise. We choose constant-time *access* for $S_{\mathcal{L}}$. This binary relation takes $2c(\log c + o(\log c) + O(1)) + o(n')$ bits of space, according to Theorem 3.1.

The total space is $c\,(\log u + \log c + \log n' + o(\log(c+n')) + O(\log\log u)) + n'(\log u + 3\log n' + o(\log n'))$. This can be up to half the space of Corollary 6.1 if $c \gg n'$, and never asymptotically larger.

The search for P proceeds just like for SLPs, paying time $O((m(m+h)\log n' + h\,occ)\log n')$ if using the most compact variant offered by Theorem 5.1. However, this will only find occurrences inside dictionary symbols: For each occurrence with offset o within symbol X_i, we look for all the positions $p_j = select_{\mathcal{C}}(X_i, j)$, for $j = 1, 2, \ldots$, and report the text position $select_1(B, p_j) + o$, within overall time $O(occ\log c)$. This includes the cases where X_i does not occur in $\mathcal{C}$, as in this case the occurrence will still appear in T and thus we can charge the search cost to it.

It remains to find the occurrences that overlap two or more entries in $\mathcal{C}$. To find each of them just once, we will find the partitions P_lP_r such that P_l is the suffix of a single entry in $\mathcal{C}$ and P_r is the prefix of a concatenation of entries in $\mathcal{C}$. Our second binary relation will let us find the positions $C_{k-1}\mathcal{C}[k\ldots]$ where P_l appears at the end of C_{k-1} and P_r at the beginning of $\mathcal{C}[k\ldots]$. We already know the lexicographical range of each P_l^{rev} within the $\mathcal{F}(X_i)^{rev}$s. We now binary search each corresponding P_r within the c suffixes starting at phrase beginnings. The content of the t-th lexicographical suffix is obtained by accessing $\mathcal{C}[S_{\mathcal{L}}[t]\ldots]$ (recall that $S_{\mathcal{L}}$ is in column order, one symbol per column) and expanding each symbol of $\mathcal{C}$ using the binary relation that represents the rules. This gives overall time $O(m(m+h)\log n'\log c)$ for the m binary searches (note the h overhead applies only to the last, partially expanded, symbol, as the rest are fully expanded). Now, given the m lexicographical ranges of the suffixes, we carry out the m range searches in the second binary relation in $O(m\log c)$ time, and extract each occurrence in $O(\log c)$ time (for this, we first obtain the column t of the pair, and then the position

$k = S_{\mathcal{L}}[t]$, as we cannot use $S_{\mathcal{L}}$ directly). To this we must add the $O(\log c)$ time to map from position $\mathcal{C}[k]$ to the corresponding position in T via bit vector B, and subtract $|P_l|$ to yield the final offset.

Overall, the search time can be written as $O((m(m+h)\log(c+n') + h\,occ)\log n' + occ \log c)$. This can be up to $O(\log c)$ times faster than Corollary 6.1 (if $c \gg n'$), and never worse.

Finally, to extract $T[l, r]$, we first use bitmap B to obtain the symbols of $\mathcal{C}$ to extract, and expand them one by one using the grammar, in overall time $O((r - l + h)\log n' + \log c)$.

Theorem 6.1. Let $T[1, u]$ be a text over alphabet $[1, \sigma]$ represented by an RSLP of n' rules and height h, and a sequence of c nonterminal symbols. Then it can be represented using $c(\log u + \log c + \log n' + o(\log(c + n')) + O(\log\log u)) + n'(\log u + 3\log n' + o(\log n'))$ bits of space. Any substring $T[l, r]$ can be extracted in time $O((r - l + h)\log n' + \log c)$, and the positions of the occurrences of a pattern $P[1, m]$ in T can be located in a fixed time $O(m(m+h)\log(n'+c)\log n')$ plus $O(h \log n' + \log c)$ time per occurrence reported.

6.1. Application to Re-Pair

Re-Pair [24] is a grammar-based compression method based on repeatedly replacing the most frequent pair of (terminal or nonterminal) symbols in the text by a new nonterminal, until the most frequent pair appears once. The result of Re-Pair compression is a set of n rules plus a sequence of c terminal or nonterminal symbols. It runs in $O(u)$ time and $O(u \log u)$ bits of space over a text $T[1, u]$ [24]. It is also possible to select the rules in a balanced fashion [38] so as to guarantee that $h = O(\log n)$.

Theorem 6.1 applies straightforwardly to Re-Pair compression, and provides a practical self-index based on grammar compression. As a proof of concept of its practicality, let us return to the genomics application mentioned in the Introduction [39, 27]. As sequencing technologies evolve rapidly, databases containing the genomes of many individuals of the same species are emerging. Their nominal sizes can easily reach the terabytes, whereas their high amount of repetitiveness offers opportunities for significant compression. In those articles the authors showed that existing self-indexes were unable to exploit this type and amount of repetitiveness, and engineered several compression methods that performed better. The best space/time tradeoff is offered by the so-called Run-Length Compressed Suffix Array (RLCSA). Compressors based on grammars, such as Re-Pair or LZ77, are able of exploiting such repetitiveness, yet no self-index based on them existed. Theorem 6.1 fills this gap, providing a self-index based on Re-Pair.

Let us consider the base case study of the performance of RLCSA, on a sequence formed by repeating 100 times a real-life 1MB sequence of ADN and introducing 0.1% of mutations (i.e., modifying a symbol) at random positions. This acts as a simplified model of a highly repetitive ADN sequence collection.

The RLCSA contains a compressed suffix array able of only counting the number of occurrences of a pattern, and a sampling needed for locating their positions. For this text, the counting part requires just 3.55% of the size of the original sequence [39]. The locating structure is investigated in the other article [27], but for this mutation rate, the classical solution of regularly sampling the whole collection is still the best choice. This solution offers a tradeoff where the locating time per occurrence is proportional to the space of the sampling, and independent of the sequence itself. Their experiments show a locating time of 11.45 μsec/occurrence when spending 33.08% of extra space (for a total of 36.63% considering the space for counting), and 55.75 μsec/occurrence when spending 8.99% (for a total of 12.54%).

A preliminary, and still amenable of several optimizations, implementation of Theorem 6.1, applied to the same text and run over a similar machine, achieves 39 μsec/occurrence for patterns of length 4–5, while requiring 8.51% of the original text size (recall that both self-indexes can replace the original collection). Thus it requires less time and space than the RLCSA. Our interpolation from the available data for the RLCSA is that, to achieve 39 μsec/occurrence, it would need 14.41% for sampling, for a total of 17.96% of space.

While far from a serious experimental comparison, this very preliminary result shows that the theoretical idea pursued in this paper has practical value: Our Re-Pair based self-index achieves a better space/time tradeoff than the best alternative self-index on a real-life problem. The result is likely to stay the same for other applications such as versioned software repositories and document databases.

6.2. Application to LZ78

Consider the LZ78-parsing [41] of a string T of length u, drawn over an alphabet Σ of size σ. The text is processed left-to-right and, at each step, a new *phrase* is produced from the longest possible prefix of the remaining text which is formed by a previous phrase plus a character. The process produces n phrases X_i, corresponding to a grammar of the form $X_i \to X_j\alpha$, where $j < i$ and $\alpha \in \Sigma$. The text is obtained by expanding the sequence $\mathcal{C} = X_1X_2 \ldots X_n$.

Much research has been carried out to obtain self-indexes for this compression method [31, 3, 36], usually called the *LZ-Index*. The first proposal [31] achieves $4n \log n + 2n \log \sigma + o(n \log n)$ bits of space, and is able of locating the *occ* occurrences of $P[1, m]$ in time $O(m^3 \log \sigma + (m + occ) \log n)$. In order to report true text positions of occurrences (and not just phrase positions), $n \log \frac{u}{n} + O(n) + o(u)$ additional bits are necessary, for a total of $n \log u + 3n \log n + 2n \log \sigma + o(u + n \log n)$ bits of space.

A verbatim application of Theorem 6.1 leads to about doubling this space. We show now that, by a slight adaptation of our general technique to the specificities of the LZ78 grammar, we can achieve a result that is competitive with the previous proposals, carefully focused on LZ78.

Let us first consider the first binary relation of Theorem 6.1. Because the right-hand side of rules is always a character, our binary relation has actually σ columns. The rules can always be ordered by $\mathcal{F}(X_l)^{rev}$ (that is, their row order), and permutation π serves as a tool to know their original identifier (which coincides with their only occurrence position in $\mathcal{C}$); π^{-1} is needed only for extracting $T[l, r]$. We do not need to store the lengths $|\mathcal{F}(X_l)|$, as we descend always to the left rule knowing that the length of the child is one less than its parent. Finally, $n' = n$ since $\mathcal{C}$ mentions only nonterminals. This makes the space $n(2 \log n + \log \sigma + o(\log n))$ bits (σ is assumed to be $o(n)$ in LZ78 self-indexes, so we do the same for comparison). The $n \log \sigma$ bits are for S_B and the $2n \log n$ for π and $S_{\mathcal{L}}$. The operations on S_B run in $O(\log \sigma)$ time and those on $S_{\mathcal{L}}$ run in $O(1)$ time for $select$ and $O(\log \log n)$ time for $access$.

Sequence $\mathcal{C} = X_1X_2 \ldots X_n$ does not need to be stored. The bitmap B is stored as in the LZ78 proposal, requiring $n \log \frac{u}{n} + O(n) + o(u)$ bits and doing the mapping in constant time [34].

Finally, for the second binary relation of Theorem 6.1, we do not require $S_{\mathcal{L}}$, as we know that the element at row i is $X_{\pi(i)}$ and thus the label is $\pi(i)+1$. Therefore the structure requires $n(\log n+o(\log n))$ bits. Furthermore, we do not need X_B because there is also one point per row in the binary relation.

Thus the total space is $n \log u + 2n \log n + n \log \sigma + o(u + n \log n)$ bits, which is less by a $n \log n + n \log \sigma$ term than the original LZ78 proposal.

The search for P starts by locating the occurrences within the nonterminals. The only possible partition of $P[1, m]$ is $P = P[1, m - 1]P[m]$. The second part is easily searched for in constant time,

whereas the first part requires $O(m \log \sigma \log n)$ time because we search the reversed rules, which are conveniently unrolled in right-to-left order. The $O(\log \sigma)$ cost is that of accessing the rules and the $O(\log n)$ corresponds to the binary search. Once the binary searches are finished, each occurrence within rules is extracted in $O(\log \sigma + \log \log n)$ time. These are mapped to $\mathcal{C}$ using π and B in constant additional time.

To spot the occurrences that span more than one phrase, we split $P = P_l P_r$ in all the $m - 1$ possible ways and search for P_l in the rows in time $O(m \log \sigma \log n)$ (using the first binary relation, which is faster), and for P_r in the columns in time $O((m \log n + h(\log \sigma + \log \log n) \log n)$ (using the first binary relation to extract the content of each phrase, so that we pay $O(\log \sigma + \log \log n)$ per symbol extracted and $O(\log n)$ per phrase expanded). Overall, the m searches add up to $O(m(m \log n + h(\log \sigma + \log \log n)) \log n)$ time, plus a negligible $O(m \log n)$ time for the range searches. Each occurrence within the ranges found is found in time $O(\log n)$.

Overall, the search time is $O(m^2 \log^2 n + mh(\log \sigma + \log \log n) \log n + occ \log n)$. This is not comparable with the original work [31], but under the usual assumption for random texts $h = O(\log_\sigma n)$, the time is usually dominated by $O(m^2 \log^2 n + occ \log n)$. This compares favorably with $O(m^3 \sigma + occ \log n)$ of the original work for long enough m. Although more recent developments [3] achieve essentially $(2 + \epsilon) n \log n$ bits of space and $O(m^2 + (m + occ) \log n)$ time, it is remarkable that we get close to such carefully engineered work with our general approach, even improving upon the original proposal.

7. Conclusions and Future Work

We have presented the first compressed indexed text representation based on Straight-Line Programs (SLPs), which are as powerful as context-free grammars. It achieves space close to that of the bare SLP representation (in many relevant cases, of the same order) and, in addition to just uncompressing, it permits extracting arbitrary substrings of the text, as well as carrying out pattern searches, in time usually sublinear on the grammar size. We also give interesting byproducts related to powerful SLP and binary relation representations.

We regard this as a foundational result on the extremely important problem of achieving self-indexes built on compression methods potentially more powerful than the current ones [32]. As such, there are many lines open to future research:

1. Our space complexity has an $n \log u$ term, which can be superlinear on the SLP size for very compressible texts. We tried hard to remove this term, for example by storing the sizes for some sampled nonterminals and computing it for the others, but did not succeed in producing a suitable sampling on the grammar DAG. The problem is related to minimum cuts in graphs [2], which is not easy.

2. We have an $O(h)$ term in the time complexities, which in case of very skewed grammar trees can be as bad as $O(n)$. There exist methods to balance a grammar to achieve $h = O(\log u)$ [37], but they introduce a space penalty factor of $O(\log u)$, which is too large in practice. It would be interesting to achieve less balancing (e.g., $h = O(\sqrt{u})$, as in LZ78) in exchange for a much lower space penalty.

3. We have an $O(m^2)$ term in the search time. It would be interesting to try to reduce it to $O(m)$, as done for some LZ78-based compressed indexes [36]. We could also use longest common prefixes (LCPs) to try to reduce an $O(m \log n)$ to $O(m+\log n)$ time in the binary search, as in suffix arrays [28]. The LCPs between successive elements in the $\mathcal{F}(X_i)$s or $\mathcal{F}(X_i)^{rev}$s could be obtained via lowest common ancestor (LCA) queries on the compressed trees [6].

4. The construction time of our index is reasonable, but the space is $O(u \log u)$ bits, which can be problematic. It should be possible to use compressed suffix trees/arrays that can be built within compressed space [26, 35] to greatly reduce the construction space in exchange for slightly higher construction time.

5. Finally, as in other compressed indexes, there is the challenge of updating the SLP and the index upon changes in the text, of working efficiently on secondary memory, and of allowing more complex searches [32] (several inspiring problems are given in recent work [17]). Extending the technique to LZ77-based compression [40] is also an interesting challenge.

Acknowledgements

We would like to thank Rossano Venturini for describing to us a simplified version of the balanced grammar construction [38] applied to Re-Pair. We would also like to thank Miguel Ángel Martínez for providing us the preliminary experimental results on the self-indexing technique applied to the Re-Pair compression algorithm.

References

[1] Amir, A., Benson, G.: Efficient two-dimensional compressed matching, *Proc. 2nd Data Compression Conference (DCC)*, 1992.

[2] Arora, S., Hazan, E., Kale, S.: $O(\sqrt{\log n})$ approximation to SPARSEST CUT $O(n^2)$ in time, *Proc. 45th Annual Symposium on Foundations of Computer Science (FOCS)*, 2004.

[3] Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index, *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, 2006.

[4] Barbay, J., Golynski, A., Munro, I., Rao, S. S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents, *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, 2006.

[5] Bender, M., Farach-Colton, M.: The level ancestor problem simplified, *Theoretical Computer Science*, **321**(1), 2004, 5–12.

[6] Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V., Rao, S. S.: Representing trees of higher degree, *Algorithmica*, **43**(4), 2005, 275–292.

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

[8] Clark, D.: *Compact Pat Trees*, Ph.D. Thesis, University of Waterloo, 1996.

[9] Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction, *Journal of the ACM*, **47**(6), 2000, 987–1011.

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

[11] Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes, *ACM Transactions on Algorithms*, **3**(2), 2007, article 20.

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

[13] Gasieniec, L., Potapov, I.: Time/space efficient compressed pattern matching, *Fundamenta Informaticae*, **56**(1-2), 2003, 137–154.

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

[15] Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes, *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[16] Gupta, A.: *Succinct Data Structures*, Ph.D. Thesis, Dept. of Computer Science, Duke University, 2007.

[17] Inenaga, S., Bannai, H.: Finding Characteristic Substrings from Compressed Texts, *Proc. 14th Prague Stringology Conference*, Czech Technical University in Prague, 2009.

[18] Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction, *Proc. 30th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 2719, 2003.

[19] Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching, *Proc. 3rd South American Workshop on String Processing (WSP)*, Carleton University Press, 1996.

[20] Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions, *Nordic Journal of Computing*, **4**(2), 1997, 172–186.

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

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

[23] Knuth, D., Morris, J., Pratt, V.: Fast pattern matching in strings, *SIAM Journal on Computing*, **6**(1), 1977, 323–350.

[24] Larsson, J., Moffat, A.: Off-line dictionary-based compression, *Proceedings of the IEEE*, **88**(11), 2000, 1722–1732.

[25] Mäkinen, V., Navarro, G.: Rank and select revisited and extended, *Theoretical Computer Science*, **387**(3), 2007, 332–347.

[26] Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes, *ACM Transactions on Algorithms*, **4**(3), 2008, article 32.

[27] Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes, *Proc. 13th Annual International Conference on Computational Molecular Biology (RECOMB)*, LNCS 5541, 2009.

[28] Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches, *SIAM Journal of Computing*, **22**(5), 1993, 935–948.

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

[30] Munro, J., Raman, R., Raman, V., Rao, S. S.: Succinct representations of permutations, *Proc. 30th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 2719, 2003.

[31] Navarro, G.: Indexing text using the Ziv-Lempel trie, *Journal of Discrete Algorithms*, **2**(1), 2004, 87–114.

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

[33] Nevill-Manning, C., Witten, I., Maulsby, D.: Compression by induction of hierarchical grammars, *Proc. 4th Data Compression Conference (DCC)*, 1994.

[34] Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[35] Russo, L., Navarro, G., Oliveira, A.: Dynamic fully-compressed suffix trees, *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, 2008.

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

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

[38] Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression, *Journal of Discrete Algorithms*, **3**, 2005, 416–430.

[39] Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-length compressed indexes are superior for highly repetitive sequence collections, *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, 2008.

[40] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, **23**(3), 1977, 337–343.

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