Indexing Straight-Line Programs*

Francisco Claude† Gonzalo Navarro‡

October 29, 2008

Abstract

Straight-line programs offer powerful text compression by representing a text $T[1,u]$ in terms of a 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 the first grammar representation able of extracting text substrings, and of searching the text for patterns, in time $o(n)$. Its size is of the same order of that of a plain SLP representation, and it can be of independent interest for other grammar-based problems. We also give some byproducts on representing binary relations.

1 Introduction and Related Work

Grammar-based compression is a well-known technique since at least the seventies [53, 50, 3, 28, 47], and still a very active area of research stimulated by the recent interest in XML compression [33, 22, 37]. The main idea is to replace a given text $T[1,u]$ by a context-free grammar (CFG) from which T can be derived. In fact, two different approaches fall under the same name [28]. In the first, there is a unique grammar $\mathcal{G}$ that can generate all the valid texts we wish to compress. Then the compressed file just indicates which rules must be applied to derive T from $\mathcal{G}$. This approach is useful when the class of texts of interest has some structure that can be exploited in order to compress them, for example program code (following the syntax of a language) or XML files that follow a DTD [8, 31, 33].

A second approach to grammar-based compression aims at deriving, from T, a specific grammar $\mathcal{G}$ that generates the single string T, and then storing $\mathcal{G}$ instead of T. Some examples are LZ78 [53], Re-Pair [32] and Sequitur [44], among many others [9]. This technique does not assume that T belongs to a class of texts with special properties, and it has been shown to be a universal compression method [28]. This is our focus in this paper.

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

*First author funded by NSERC of Canada and Go-Bell Scholoarships Program. Second author funded in part by Fondecyt Grant 1-080019, and by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

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

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

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 mentioned in the previous paragraph can be straightforwardly translated, with no significant penalty, into SLPs. SLPs are in practice competitive with the best compression methods [18].

There are textual substitution compression methods which are more powerful than those CFG-based. *Collage Systems* [27] is a formalism that encompasses several operations to describe strings. SLPs (and CFGs) are equivalent to a subclass called *regular collage systems* in the taxonomy. More general collage systems include operations like prefix/suffix truncation and repetitions, which allow them to express schemes like LZ77 [52], that 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 [48].

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 [48, 9]. Moreover, some popular grammar-based compressors such as LZ78, Re-Pair and Sequitur, can generate a compressed file much larger than the smallest SLP [9]. Yet, a simple method to achieve an $O(\log u)$-approximation is to parse T using LZ77 and then converting it into an SLP [48]. This has the additional advantage that the SLP 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.)

Given a compressed text $T[1,u]$, one could aim at extracting only a portion of it, or at determining the presence of substrings $P[1,m]$ in it, and even see a context around each occurrence, all without decompressing T (which would require $O(u)$ time overall). This would permit knowing whether it is worth to decompress T. *Compressed pattern matching* [1] is the problem of finding the occurrences of P in T from a compressed representation of T. *Fully compressed pattern matching* is the variant where P is compressed as well. After much research by the community on specific compression methods [16, 2, 35, 26, 12, 29, 18, 43], Kida et al. [27] showed that, for general SLP (and CFG) compression, compressed pattern matching can be carried out in $O(n + m^2 + occ)$ time to output the *occ* occurrences. Fully compressed pattern matching requires time $O(n^2 + mn \log n)$, where here m is the size of the SLP representing P (the occurrences are represented in a compact form) [23]. Some slightly worse results were obtained for more general collage systems.

Despite there has been a substantial amount of work on pattern matching on text compressed using different types of textual substitution compression methods, we note that all those approaches refer to *sequential* pattern matching, that is, one has to traverse the whole compressed text. In this paper we focus on *indexed* searching, which is the only practical choice when managing large compressed text collections. In indexed text searching, some data structures are built on the text so as to permit searching in sublinear time ($o(n)$ in our case). A related issue is that of efficient random access to a grammar-based compressed file, without having to decompress the whole of it [17].

In fact, there has been much work on compressing and indexing natural language text collections [51, 42] and also general texts [41], but not based on grammar-based compression. The only

exception are those based on LZ78-like compression [40, 14, 46]. These are *self-indexes*, meaning that the compressed text representation itself is powerful enough to support indexed searches. Recently, several self-indexes were tested on a scenario of highly repetitive text collections, modeling a genomics application [49], concluding that none of the existing self-indexes was able to capture these redundancies. Even the LZ78-based ones failed, which is not surprising given that LZ78 can output a text exponentially larger than the smallest SLP. This type of application claims for self-indexes based on stronger compression methods, such as general SLPs.

In this paper we introduce the *first* SLP representation that allows for (a) extracting any substring of T, and (b) obtaining the positions of the occurrences of an uncompressed pattern P in T. 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 extract any substring $T[l, r]$ in time $O((h + r - l) \log n)$, where h is the height of the derivation tree. It can output *occ* occurrences of $P[1, m]$ in time $O((m(m + h) + h\, occ) \log n)$ (see the detailed results in Thm. 5). 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 which in addition permits a kind of range queries. These results are of independent interest.

In practical terms, 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 (such as biological sequences or versioned documents), by combining good compression and efficient indexed searching. In theoretical terms, ours is the first result on representing an SLP in such a way that searching in time sublinear in the SLP size is possible.

2 Basic Concepts

2.1 Succinct 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:

- $rank_S(a, i)$ 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 [10]. 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 [45].

The general case has been proved to be a little harder. Wavelet trees [20] 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

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

[15] with multiary wavelet trees to achieve $O(1 + \frac{\log \sigma}{\log \log n})$ time within the same space. Another interesting proposal [19], focused on large alphabets, achieves $n \log \sigma + no(\log \sigma)$ 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 [19] 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*.

Now we describe the wavelet tree, and its extension to support range queries. 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 [34] showed how to use a wavelet tree to represent a permutation π of $[1, n]$ so as to answer *range queries*. We give here an almost identical version 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 $S[i_1, i_2]$ $(1 \leq i_1 \leq i_2 \leq n)$ which are in the range $[j_1, j_2]$ $(1 \leq j_1 \leq j_2 \leq \sigma)$. The operation takes $O(\log \sigma)$ to count the number of results [34], 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 $[1, \sigma]$ [34].

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** $i_2 - i_1 + 1$
$tm \leftarrow \lfloor (t_1 + t_2)/2 \rfloor$
$[i_{l1}, i_{l2}] \leftarrow [rank_{B_v}(0, i_1 - 1) + 1, rank_{B_v}(0, i_2)]$
$[i_{r1}, i_{r2}] \leftarrow [rank_{B_v}(1, i_1 - 1) + 1, rank_{B_v}(1, i_2)]$
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 the wavelet tree node, B_v the bitmap stored at v, and v_l/v_r its left/right children. It is invoked with RANGE$(root, [i_1, i_2], [j_1, j_2], [1, \sigma])$.

2.2 Straight-Line Programs

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

Definition 1 *[26] 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 *[48] 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 $height(\mathcal{G}) = height(X_n)$.*

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 1 *[48] 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 [9], it is interesting that one can find in linear time a reasonable (and balanced) approximation.

Theorem 2 *[48] 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 $|\mathcal{L}(l)|$.

We build on an idea by Barbay et al. [5]. We define, for $a \in A$, $s(a) = b_1 b_2 \ldots b_k$, where $b_i < b_{i+1}$ for $1 \leq i < k$ and $B(a) = \{b_1, b_2, \ldots, b_k\}$. We build a string $S_B = s(1)s(2) \ldots s(n_1)$ and write down the cardinality of each $B(a)$ in unary on a bitmap $X_B = 0^{|B(1)|}10^{|B(2)|}1 \ldots 0^{|B(n_1)|}1$. Another sequence $S_{\mathcal{L}}$ lists the labels $\mathcal{L}(a, b)$ in the same order they appear in S_B: $S_{\mathcal{L}} = l(1)l(2) \ldots l(n_1)$, $l(a) = \mathcal{L}(a, b_1)\mathcal{L}(a, b_2) \ldots \mathcal{L}(a, b_k)$. Table 1 shows an example. We also store a bitmap $X_A = 0^{|A(1)|}10^{|A(2)|}1 \ldots 0^{|A(n_2)|}1$.

We represent S_B using wavelet trees [20], $\mathcal{L}$ with the structure for large alphabets [19], and X_A and X_B in compressed form [45]. 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:

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

Table 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.

- $|A(b)|$: This is just $select_{X_A}(1, b) - select_{X_A}(1, b-1) - 1$.
- $|B(a)|$: It is computed in the same way using X_B.
- $\mathcal{L}(a, b)$: Compute $y \leftarrow select_{X_B}(1, a-1) - a + 1$. Now, if $rank_{S_B}(b, y) = rank_{S_B}(b, y + |B(a)|)$ then we know a and b are not related and return $\perp$. Otherwise, we return $S_{\mathcal{L}}[select_{S_B}(b, rank_{S_B}(b, y + |B(a)|))]$.
- $A(b)$: We first compute $|A(b)|$ and then retrieve the i-th element by doing $y_i \leftarrow select_{S_B}(b, i)$ and returning $1 + select_{X_B}(0, y_i) - y_i$.
- $B(a)$: This is $S_B[select_{X_B}(1, a-1) - a + 2 \ldots select_{X_B}(1, a) - a]$.
- $\mathcal{R}(a_1, a_2, b_1, b_2)$: We first determine which elements in S_B correspond to the range $[a_1, a_2]$. We set $a'_1 \leftarrow select_{X_B}(1, a_1 - 1) - a_1 + 2$ and $a'_2 \leftarrow select_{X_B}(1, a_2) - a_2$. Then, using range queries in a wavelet tree [34] (recall Algorithm 1), we 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 of l in $S_{\mathcal{L}}$. For the i-th occurrence we find $y_i \leftarrow select_{S_{\mathcal{L}}}(l, i)$, then we compute $b \leftarrow S_B[y_i]$ and $a \leftarrow 1 + select_{X_B}(0, y_i) - y_i$. 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 [19] for S_B.

Theorem 3 *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 permits various queries on the SLP within essentially the same space of a plain representation.

Let us assume for simplicity that all the symbols in Σ are used in the SLP, and thus $\sigma \leq n$ is the effective alphabet size. If this is not the case and $\max(\Sigma) = \sigma' > n$, we can always use a mapping $S[1, \sigma']$ from Σ to the effective alphabet range $[1, \sigma]$, using $rank$ and $select$ in S. By using Raman et al.'s representation [45], S requires $O(\sigma \log \frac{\sigma'}{\sigma}) = O(n \log \frac{\sigma'}{n})$ bits. Any representation of such an SLP would need to pay for this space.

A plain representation of an SLP with n rules 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 Thm. 3, 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.

Recall that the SLP has n rules of the form $X_i \to X_l X_r$ or $X_i \to \alpha$. We first transform the SLP into an equivalent one enforcing two simple conditions: (1) there are no repeated right hand sides in the rules, (2) rules of the form $X_i \to \alpha$ are ordered, that is, if $X_i \to \alpha_1$ and $X_j \to \alpha_2$, then $i < j$ iff $\alpha_1 < \alpha_2$. Both are easily enforced by renaming rules.

In our binary relation representation, every row represents a symbol X_l and every column a symbol X_r. 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 $\alpha \in \Sigma$. We also store a bitmap $C[1, \sigma]$, where we set $C[\alpha] = 1$ for those α. Thus we know $X_i \to \alpha$ in constant time because $Y[i] = 1$ and $\alpha = select_C(1, rank_Y(1, i))$. The total space is $(n + \sigma)(1 + o(1)) = O(n)$ bits [10].

This representation lets us answer 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 constant time using $i = select_Y(1, rank_C(1,\alpha))$ (if $C[\alpha] = 0$, there is no such X_i).

- *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 $\{\mathcal{L}(i,r), r \in B(j)\}$ and the second using $\{\mathcal{L}(l,i), l \in A(j)\}$, in $O(\log n)$ time per each X_i 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 [19] (alternative (i) in Thm. 3), to achieve $O(\log\log n)$ time for all the other queries.

Theorem 4 *An SLP $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma)$, $\Sigma = [1, \sigma]$, $\sigma \le n$, can be represented using $2n \log n + o(n \log n)$ 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 $O(\log\log n)$. For arbitrary integer Σ one needs additional $O(n \log \frac{\max(\Sigma)}{n})$ bits.*

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 effective alphabet $\Sigma = [1,\sigma]$, can be represented with an SLP of n rules.

Definition 3 *A* Grammar-Compressed Text (GCT) *$\mathcal{G} = (X, \Sigma, s)$ is a grammar with nonterminals $X = \{X_1, X_2, \ldots, X_n\}$, terminals Σ, and two types of rules: (i) $X_i \to \alpha$, where $\alpha \in \Sigma$, (ii) $X_i \to X_l X_r$, such that:*

1. *The X_is can be renumbered X'_i in order to obtain an SLP.*
2. *$\mathcal{F}(X_i) \preceq \mathcal{F}(X_{i+1}), 1 \le i < n$, being $\preceq$ the lexicographical order.*
3. *There are no duplicate right hands in the rules.*
4. *X_s is mapped to X'_n, so that $\mathcal{G}$ represents the text $T = \mathcal{F}(X_s)$.*

It is clear that every SLP can be transformed into a GCT, by removing duplicates and lexicographically sorting the expanded phrases.

We will represent a GCT $\mathcal{G}$ using a variant of Thm. 4. The rows will represent X_l as before, but these will be sorted by *reverse* lexicographic order, as if they represented $\mathcal{F}(X_l)^{rev}$. The columns will represent X_r, ordered lexicographically by $\mathcal{F}(X_r)$. We will also store a permutation π_R, which maps reverse to direct lexicographic ordering. This must be used to translate row positions to

nonterminal identifiers. We use Munro et al.'s representation [39] for π_R, with parameter $\epsilon = \frac{1}{\log n}$, so that π_R can be computed in constant time and π_R^{-1} in $O(\log n)$ time, and the structure needs $n \log n + O(n)$ bits of space.

With the SLP representation and π_R, the space for the GCT is $3n \log n + o(n \log n)$ bits. We add other $n \log u$ bits for storing the lengths $|\mathcal{F}(X_i)|$ for all the nonterminals X_i.

5.1 Extraction of Text from a GCT

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 $X_i \to X_l X_r$, 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. 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 $O((h + j' - j) \log n)$, where $h = height(\mathcal{G})$. 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 a GCT

Our 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 a GCT of n rules. As in previous work [25], except for the special case $m = 1$, occurrences can be divided into *primary* and *secondary*. A primary occurrence in $\mathcal{F}(X_i)$, $X_i \to X_l X_r$, 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 primary occurrence is at $X_i \to \alpha$ 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 GCT 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*.

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 $\pi_R^{-1}(i)$. 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 $\pi_R(l)$ is obtained from the result using an *access to rule* query. 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 $O((m + h\,occ)\log n)$, where $h = height(\mathcal{G})$.

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 counting the primary occurrences, and it corresponds to $occ = 0$. The remaining problem is 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. We do the first binary search in the nonterminals as they are ordered in the GCT. In order to do the string comparisons, we extract the first m terminals of $\mathcal{F}(X_i)$, in time $O((m+h)\log n)$ (Sec. 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_lP_r. The search within the reverse phrases is similar, except that we extract the m rightmost terminals and must use π_R 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 [38] 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). By using the cardinal tree representation of Benoit et al. [7] 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 three for the $\mathcal{F}(X_i)^{rev}$s, to $X_{\pi_R(i)}$). 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 [7].

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 [11]. This can be obtained in time $O((\ell + h)\log n)$, where ℓ is the skip value (Sec. 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 *Let $T[1,u]$ be a text over an effective alphabet $[1,\sigma]$ represented by an SLP of n rules and height h. Then there exists a representation of T using $n(\log u + 3\log n + O(\log\sigma + \log h) + o(\log n))$ bits, such that any substring $T[l,r]$ can be extracted in time $O((r-l+h)\log n)$, and the positions of occ occurrences of a pattern $P[1,m]$ in T can be found in time $O((m(m+h)+h\,occ)\log n)$. By removing the $O(\log h)$ term in the space, search time raises to $O((m^2+occ)h\log n)$. By further removing the $O(\log\sigma)$ term in the space, search time raises to $O((m(m+h)\log n + h\,occ)\log n)$. The existence problem is solved within the time corresponding to $occ=0$.*

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 anyway extremely small compared to the *original* text size.

5.4 Construction

We have not discussed construction times for our index (given the SLP). Those are $O(n\log n)$ for the binary relation part, and all the lengths $|\mathcal{F}(X_i)|$ could be easily obtained in $O(n)$ time. Sorting the strings lexicographically, as well as constructing the tries, however, can take as much as $\sum_{i=1}^{n}|\mathcal{F}(X_i)|$, which can be even $\omega(u)$. Yet, as all the phrases are substrings of $T[1,u]$, we can build the suffix array of T in $O(u)$ time [24], record one starting text position of each $\mathcal{F}(X_i)$ (obtained by expanding T from the grammar), and then sorting them in $O(n\log n)$ time using the inverse suffix array permutation (the ordering when one phrase is a prefix of the other is not relevant for our algorithm). To build the Patricia trees we can build the suffix tree in $O(u)$ time [13], mark the n suffix tree leaves corresponding to phrase beginnings, prune the tree to the ancestors of those leaves (which are $O(n)$ after removing unary paths again), and create new leaves with the corresponding string depths $|\mathcal{F}(X_i)|$. The point to insert the new leaves are found by binary searching the string depths $|\mathcal{F}(X_i)|$ with level ancestor queries [6] from the suffix tree leaves. The process takes $O(u+n\log n)$ time and $O(u\log u)$ bits of space. Reverse phrases are handled identically.

5.5 Faster Locating

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 $O(h\,occ\log n)$ term 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. Separately, we store a balanced binary search tree that, given i, gives the memory position of X_i, if it exists (and it permits freeing all the memory areas at the end).

Now, we repeatedly extract an element X_i with smallest $|\mathcal{F}(X_i)|$ from H, we find using our binary relation data structures all the others X_j 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 internal 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. 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 positions must overlap, and 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 in $O(m)$ time we can obtain those occurrences in constant time each: Let $last - first = d$, so $last - d$ is the *first* primary occurrence. This means that $P[d+1, m] = P[1, m-d]$. Assume there exists $d' < d$ such that $P[d'+1, m] = P[1, m-d']$; then $last - d'$ is also a primary occurrence, and vice versa. If we know all the d' values for which the condition on P holds, and store a table $O[1, m]$ with $O[d]$ storing the largest valid $d' < d$, 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 $d = 0$, that is, we have reported *last*.

Table $O[1, m]$ can be computed in $O(m)$ time using a variant of KMP algorithm [30], which searches $P\$$ for P (where \$ is a special symbol not ocurring in P) and keeps track of the positions where P is aligned when the "text" pointer is at the "\$" (those are called the "borders" of P).

Let us now analyze the algorithm. Although each primary 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 $X_i \to X_l X_r$ 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 $O(\min(h\,occ, n)\log n + occ)$, provided we can afford the extra space at search time.

6 Application to Re-Pair

Re-Pair [32] 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 *dictionary* $\mathcal{D}$ of d rules plus a *sequence* $\mathcal{C}$ of n (terminal or nonterminal) symbols.

As such, we can handle Re-Pair by adding $n-1$ rules to the dictionary, so that the initial symbol of the resulting SLP produces the sequence. We must also create σ rules that produce the terminals. The resulting SLP has $n' = \sigma + d + n - 1$ rules and, if we create those $n-1$ rules in balanced fashion, it will have height $h' = h + \log n$, where h is the maximum rule height in the dictionary. Thus Theorem 5 can handle it using, for example $(n+d+\sigma)(\log u + 3\log(n+d+\sigma) + o(\log(n+d+\sigma)))$ bits, and searching in time $O((m(m+h+\log n)\log(n+d+\sigma) + (h+\log n)\,occ)\log(n+d+\sigma))$.

We show next that this can be improved by specializing our solution a bit. Our data structures will be as follows:

1. We use our binary relation data structure to represent just the dictionary, which is a *forest*, that is, a set of trees. We add the σ rules to generate the terminals. Thus it will require $(d+\sigma)(\log u + 3\log(d+\sigma) + o(\log(d+\sigma)))$ bits of space, according to Theorem 3.

2. The sequence $\mathcal{C}$ is be represented with the structure for sequences over large alphabets [19]. This will require $n(\log(d+\sigma) + o(\log(d+\sigma))$ bits of space and carry out *access* in $O(\log\log(d+\sigma))$ 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 [21, Thm. 17 p. 153] such that it uses $n\log\frac{u}{n} + O(n\log\log\frac{u}{n})$ bits and supports *rank* and *select* in $O(\log n)$ time.

4. We store another labeled binary relation of $(d+\sigma)$ rows and n columns. Value $1 \le i \le d+\sigma$ is related to $1 \le j \le n$ with label $2 \le k \le n$ 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 $\mathcal{F}(\mathcal{C}[k-1])^{rev}$, among all the distinct reversed nonterminals (and terminals), 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 in practice bitmaps X_A and X_B. We choose constant-time *access* for $S_{\mathcal{L}}$. This binary relation takes $2n(\log n + o(\log n))$ bits of space, according to Theorem 3.

Thus the total space is $n(\log u + \log n + \log(d+\sigma)) + o(\log n + \log(d+\sigma)) + O(\log\log \frac{u}{n})) + (d+\sigma)(\log u + 3\log(d+\sigma) + o(\log(d+\sigma)))$. This can be up to half the space of the trivial solution if $n >> d+\sigma$, and in no case can be asymptotically larger.

The search for P proceeds just like for SLPs, paying time $O((m(m+h)\log(d+\sigma)+h\,occ)\log(d+\sigma))$. 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 $O(occ \log n)$.

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 partition $P_l P_r$ such that P_l is the suffix of a single entry in $\mathcal{C}$ and P_r is the prefix of a suffix in $\mathcal{C}$. 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 n suffixes starting at phrase beginnings. The contents of the t-th lexicographical suffix is obtained by accessing $\mathcal{C}[S_{\mathcal{L}}[t]\ldots]$ and expanding each symbol of $\mathcal{C}$ using the dictionary representation. This gives overall time $O(m(m+h)\log(d+\sigma)\log n)$ for the m binary searches. Now given the m lexicographical ranges of the suffixes, we carry out the m range searches in $O(m \log n)$ time, and extract each occurrence in $O(\log n)$ time. To this we must add the $O(\log n)$ time to map from position in $\mathcal{C}$ to position in T via bit vector B.

Overall, the search time can be written as $O((m(m+h)\log(n+d+\sigma)+h\,occ)\log(d+\sigma))+occ\log n$. This can be up to $O(\log n)$ times faster than the trivial approach (if $n >> d+\sigma$), and never worse.

7 Conclusions and Future Work

We have presented the first indexed compressed text representation based on Straight-Line Programs (SLP), 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 [41]. 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 [4], 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 $h = O(\log u)$ [48], but it introduces a space penalty factor of $O(\log u)$, which is too large in practice. It

would be interesting to achieve less balancing (e.g., $h = O(\sqrt{u})$ 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 LZ78-based compressed indexes [46]. We could also use longest common prefixes (LCPs) to try to reduce an $O(m \log n)$ to $O(m + \log n)$ in the binary search [36]. 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 [7].

4. 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 [41]. Extending the technique to LZ77-based compression [52] is also an interesting challenge.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 2nd DCC*, pages 279–288, 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comp. Sys. Sci.*, 52(2):299–307, 1996.

[3] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proc. 8th DCC*, pages 119–128, 1998.

[4] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\log n})$ approximation to SPARSEST CUT $O(n^2)$ in time. In *Proc. 45th FOCS*, pages 238–247, 2004.

[5] J. Barbay, A. Golynski, I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th CPM*, LNCS 4009, pages 24–35, 2006.

[6] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comp. Sci.*, 321(1):5–12, 2004.

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

[8] R. Cameron. Source encoding using syntactic information source models. *IEEE Trans. Inf. Theory*, 34:843–850, 1988.

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

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

[11] D. Clark and I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[12] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.

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

[14] P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.

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

[16] L. Gasieniec, M. Karpinksi, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encodings. In *Proc. 5th SWAT*, pages 392–403, 1996.

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

[18] L. Gasieniec and I. Potapov. Time/space efficient compressed pattern matching. *Fund. Inf.*, 56(1-2):137–154, 2003.

[19] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.

[20] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[21] A. Gupta. *Succinct Data Structures*. PhD thesis, Dept. of Computer Science, Duke University, 2007.

[22] S. Harrusi, A. Averbuch, and A. Yehudai. XML syntax conscious compression. In *Proc. 16th DCC*, pages 402–411, 2006.

[23] S. Inenaga, A. Shinohara, and M. Takeda. A fully compressed pattern matching algorithm for simple collage systems. *Int. J. Found. Comp. Sci.*, 16(6):1155–1166, 2005.

[24] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th ICALP*, LNCS v. 2719, pages 943–955, 2003.

[25] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd WSP*, pages 141–155. Carleton University Press, 1996.

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

[27] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comp. Sci.*, 298(1):253–272, 2003.

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

[29] S. Klein and D. Shapira. Pattern matching in Huffman encoded texts. In *Proc. 11th DCC*, pages 449–458, 2001.

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

[31] E. Kourapova and B. Ryabko. Application of formal grammars for encoding information sources. *Problof Inf. Transm.*, 31:23–26, 1995.

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

[33] G. Leighton, J. Diamond, and T. Müldner. AXECHOP: a grammar-based compressor for XML. In *Proc. 15th DCC*, page 467, 2005.

[34] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comp. Sci.*, 387(3):332–347, 2007.

[35] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inf. Sys.*, 15(2):124–136, 1997.

[36] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.

[37] S. Maneth, N. Mihaylov, and S. Sakr. XML tree structure compression. In *Proc. 19th DEXA Workshops*, pages 243–247, 2008.

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

[39] J. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, LNCS 2719, pages 345–356, 2003.

[40] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discr. Alg.*, 2(1):87–114, 2004.

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

[42] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retr.*, 3(1):49–77, 2000.

[43] G. Navarro and M. Raffinot. Practical and flexible pattern matching over ziv-lempel compressed text. *J. Discr. Alg.*, 2(3):347–371, 2004.

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

[45] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.

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

[47] W. Rytter. Compressed and fully-compressed pattern matching in one and two dimensions. *Proc. IEEE*, 88(11):1769–1778, 2000.

[48] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comp. Sci.*, 302(1-3):211–222, 2003.

[49] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th SPIRE*, LNCS, 2008.

[50] J. Storer and T. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[51] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.

[52] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

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