Self-Indexed Text Compression using Straight-Line Programs

Francisco Claude[1] * and Gonzalo Navarro[2] **

[1] David R. Cheriton School of Computer Science, University of Waterloo. fclaude@cs.uwaterloo.ca.
[2] 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.

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 $T[1,u]$ 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 [18]. Some examples are LZ78 [31], Re-Pair [19] and Sequitur [25], among many others [5].

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 [11].

There are textual substitution compression methods which are more powerful than those CFG-based [17]. A well-known one is LZ77 [30], which cannot be directly expressed using CFGs. Yet, an LZ77 parsing can be converted into an

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

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 [28].

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 [28, 5]. Moreover, some popular grammar-based compressors such as LZ78, Re-Pair and Sequitur, can generate a compressed file much larger than the smallest SLP [5]. Yet, a simple method to achieve an $O(\log u)$-approximation is to parse T using LZ77 and then converting it into an SLP [28], 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 to be practical for large databases.

There has been some work on random access to grammar-based compressed text, without decompressing all of it [10]. 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. [17], 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. Large compressed text databases require *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 $O(m\text{polylog}(n))$ time) [24], 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 [23, 8, 27]. 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 [29] 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 [29] considers a set of r genomes of length n, 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 [24, 13, 9] occupy essentially nrH_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 $nH_k + O((r+s)\log n)$ bits, that is, the compressed base sequence plus $O(\log n)$ bits per edit and per sequence. A properly designed SLP can achieve $nH_k + O(r \log n) + O(s \log^2 n)$ bits, which is much better than the current techniques. It is not as good as LZ77, self-indexes based on LZ77 are extremely challenging and do not exist yet.

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[3], 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 Thm. 3). 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 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.

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 $S[1, i]$, $rank_S(a, 0) = 0$; and $select_S(a, i)$ 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 [6]. 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 [26].

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

The general case has been proved to be a little harder. Wavelet trees [13] achieve $n \log \sigma + o(n) \log \sigma$ bits of space while answering all the queries in $O(\log \sigma)$ time. Another interesting proposal [12], 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 [12] 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*.

Mäkinen and Navarro [20] showed how to use a wavelet tree to represent a permutation π of $[1, n]$ so as to answer *range queries*. We use a trivial variant 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, and can 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 $[1, \sigma]$. The algorithms are almost identical to those for permutations [20].

2.2 Straight-Line Programs

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

Definition 1. *[16] 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. *[28] 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 that an SLP $\mathcal{G}$ of n rules generating $T[1, u]$ can be converted into a $\mathcal{G}'$ of $O(n \log u)$ rules and $height(\mathcal{G}') = O(\log u)$, in $O(n \log u)$ time [28]. Also, as several grammar-compression methods are far from optimal [5], it is interesting that one can find in linear time an $O(\log u)$ approximation to the smallest grammar, which in addition is balanced (height $O(\log u)$) [28].

3 Labeled Binary Relations with Range Queries

In this section we introduce a data structure for labeled binary relations supporting range queries. 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 $\mathcal{L} : A \times B \to L \cup \{\perp\}$, mapping pairs in $\mathcal{R}$ to labels in $L = \{1, 2, \ldots, \ell\}, \ell \geq 1$, and the others to $\perp$. We support the following queries:

- $\mathcal{L}(a,b)$.
- $A(b) = \{a,\ (a,b) \in \mathcal{R}\}$.
- $B(a) = \{b,\ (a,b) \in \mathcal{R}\}$.
- $R(a_1,a_2,b_1,b_2) = \{(a,b) \in \mathcal{R},\ a_1 \le a \le a_2,\ b_1 \le b \le b_2\}$.
- $\mathcal{L}(l) = \{(a,b) \in \mathcal{R},\ \mathcal{L}(a,b) = l\}$.
- 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. [2]. We define, for $a \in A$, $s(a) = b_1 b_2 \ldots b_k$, where $b_i < b_{i+1}$ for $1 \le i < k$ 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: $S_{\mathcal{L}} = l(1)l(2)\ldots l(n_1)$, $l(a) = \mathcal{L}(a,b_1)\mathcal{L}(a,b_2)\ldots \mathcal{L}(a,b_k)$. 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 [13], $\mathcal{L}$ with the structure for large alphabets [12], and X_A and X_B in compressed form [26]. 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:

- $|A(b)|$: 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 a and b are not related and we 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 [20], 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 [12] for S_B.

Theorem 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 $L = \{1,2,\ldots,\ell\}, \ell \ge 1$, 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 [26], 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. 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.

Definition 3. *A* Lexicographic Straight-Line Program (LSLP) $\mathcal{G} = (X, \Sigma, s)$ *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_i*s can be renumbered* X'_i *in order to obtain an SLP.*
2. $\mathcal{F}(X_i) \preceq \mathcal{F}(X_{i+1}), 1 \leq i < n$*, 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 an LSLP, by removing duplicates and lexicographically sorting the expanded phrases. We will use LSLPs in place of SLPs from now on.

Let us regard 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 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. Note 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 [6]. This works because the rules are lexicographically sorted and all the symbols in Σ are used.

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 $O(1)$ 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 $\{\mathcal{L}(i, r), r \in B(j)\}$ 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 $l_1 \leq l_2, r_1 \leq r_2$, find those i such that $X_i \to X_l X_r$ for any $l_1 \leq l \leq l_2$ and $r_1 \leq r \leq r_2$. 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 [12] (alternative (i) in Thm. 1), to achieve $O(\log \log n)$ time for all the other queries.

Theorem 2. *An SLP $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma)$, $\Sigma = [1, \sigma]$, $\sigma \leq n$, 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 LSLP-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 of n rules.

We will represent an LSLP $\mathcal{G}$ using a variant of Thm. 2. 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 [22] 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 LSLP representation and π_R, the space required 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 an LSLP

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 $O(\mathit{height}(X_i) \log n)$ 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 $O((\mathit{height}(X_i) + j' - j) \log n)$ 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 LSLP

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 an LSLP of n rules. As in previous work [15], 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 LSLP 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 $O((m + h\,occ)\log n)$.

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 LSLP. 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 $P_l P_r$. 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 [21] 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. [4] 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 [4].

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 [6]. 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 3. *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 still much smaller than the *original* text.

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 [14], 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 [7], 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 [3] 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.

6 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 [24]. As such, there are several possible improvements we plan to work on, such as (1) reducing the $n \log u$ space term; (2) reduce the $O(m^2)$ term in search times; (3) alleviate the $O(h)$ term in search times by restricting the grammar height while retaining good compression; (4) report occurrences faster than one-by-one. We also plan to implement the structure to achieve strong indexes for very repetitive text collections.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 2nd DCC*, pages 279–288, 1992.
2. 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.
3. M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comp. Sci.*, 321(1):5–12, 2004.
4. 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.
5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE TIT*, 51(7):2554–2576, 2005.
6. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.
7. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
8. P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.

9. 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.
10. L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.
11. L. Gasieniec and I. Potapov. Time/space efficient compressed pattern matching. *Fund. Inf.*, 56(1-2):137–154, 2003.
12. 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.
13. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
14. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th ICALP*, LNCS v. 2719, pages 943–955, 2003.
15. 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.
16. 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.
17. 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.
18. J. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE TIT*, 46(3):737–754, 2000.
19. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.
20. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comp. Sci.*, 387(3):332–347, 2007.
21. D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
22. J. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, LNCS 2719, pages 345–356, 2003.
23. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discr. Alg.*, 2(1):87–114, 2004.
24. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
25. C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *Proc. 4th DCC*, pages 244–253, 1994.
26. 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.
27. L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Inf. Retr.*, 11(4):359–388, 2008.
28. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comp. Sci.*, 302(1-3):211–222, 2003.
29. 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 5280, pages 164–175, 2008.
30. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TIT*, 23(3):337–343, 1977.
31. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE TIT*, 24(5):530–536, 1978.