Balancing Run-Length Straight-Line Programs*

Gonzalo Navarro, Francisco Olivares, and Cristian Urbina

CeBiB — Center for Biotechnology and Bioengineering
Departament of Computer Science, University of Chile

Abstract. It was recently proved that any SLP generating a given string w can be transformed in linear time into an equivalent balanced SLP of the same asymptotic size. We show that this result also holds for RLSLPs, which are SLPs extended with run-length rules of the form $A \to B^t$ for $t > 2$, deriving $\texttt{exp}(A) = \texttt{exp}(B)^t$. An immediate consequence is the simplification of the algorithm for extracting substrings of an RLSLP-compressed string. We also show that several problems like answering RMQs and computing Karp-Rabin fingerprints on substrings can be solved in $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ time, g_{rl} being the size of the smallest RLSLP generating the string, of length n. We extend the result to solving more general operations on string ranges, in $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ applications of the operation. In general, the smallest RLSLP can be asymptotically smaller than the smallest SLP by up to an $\mathcal{O}(\log n)$ factor, so our results can make a difference in terms of the space needed for computing these operations efficiently for some string families.

Keywords: Run-length straight-line programs · Substring range problems · Repetitive strings

1 Introduction

Enormous collections of data are being generated at every second nowadays. Already storing this data is becoming a relevant and practical challenge. Compression serves to represent the data within reduced space. Still, just storing the data in compressed form is not sufficient in many cases; one also requires to construct data structures that support various queries within the compressed space. For example, *index* data structures support the search for short patterns in compressed strings. In areas like Bioinformatics, these collections of strings are often very repetitive [22], which makes traditional compressors and indexes based on Shannon's entropy unsuitable for this task [19].

Over the years, several compressors and data structures exploiting repetitiveness have been devised. Examples of this are the Lempel-Ziv family [18,16] and the run-length Burrows-Wheeler transform (BWT) [3,8]. While compressors based on Lempel-Ziv achieve the best compression ratios, indexes based on them are not very fast and provide limited functionality. On the other hand, indexes

* Funded in part by Basal Funds FB0001, Fondecyt Grant 1-200038, and two Conicyt Doctoral Scholarships, ANID, Chile.

based on the BWT can efficiently solve a variety of queries over strings, but their compression ratio is far from optimal for repetitive sequences [13].

Somewhere in between of Lempel-Ziv and BWT compression is *grammar compression*. This approach consists in constructing a deterministic context-free grammar generating only the string to be compressed; such grammars are called straight-line programs (SLPs). Although finding the smallest SLP generating a string is NP-complete [4], there exist several heuristics [17,20] and approximations [11,23] producing SLPs of small size. The popularity of SLPs probably comes from their simplicity to expose repetitive patterns on strings, which is useful to avoid redundant computation in compressed space [15,25]. This makes SLPs ideal for indexing and answering queries in compressed space [2,10].

A problem that complicates such computations is that the parse tree of the grammars can be arbitrarily tall. While tasks like accessing a symbol of the string in time proportional to the parse tree height is almost trivial, achieving $\mathcal{O}(\log n)$ time on general grammars requires much more sophistication [2]. Recently, Ganardi et al. [10] showed that any SLP can be balanced without paying an (asymptotic) increase in its size. This simplified several problems that were difficult for general SLPs, but easy if the depth of their parse tree is $\mathcal{O}(\log n)$. Accessing a symbol in time $\mathcal{O}(\log n)$ is nearly optimal, actually [24].

An extended grammar compression mechanism are the run-length SLPs introduced by Nishimoto et al. [21]. An RLSLPs is an SLP extended with run-length rules of the form $A \to B^t$ for some $t > 2$, which derive $\mathtt{exp}(A) = \mathtt{exp}(B)^t$. While the size of the smallest SLP generating a string of length n is always $\Omega(\log n)$, the smallest RLSLP can be of size $\mathcal{O}(1)$ for some string families, which exhibit a logarithmic gap between the compression power of SLPs and RLSLPs. RLSLP have recently gained popularity for indexing. For example, all known *locally consistent grammars* are RLSLPs, and they have been a key component in the most recent indices for repetitive text collections. A locally consistent grammar is built through consecutive applications of a locally consistent parsing, which is a method to partition a string into non-overlapping blocks, such that equal substrings are equally parsed with the possible exception of their margins. Gagie et al. [9] built an index based on locally consistent grammars using $\mathcal{O}(r \log \log n)$ space, with which they were able to count the *occ* occurrences of a length-m pattern in optimal time $\mathcal{O}(m)$ and locate them in optimal time $\mathcal{O}(m + occ)$, where r is the number of runs in the BWT [3] of the string. Kociumaka et al. [5] also built a locally consistent grammar to index a string. Their grammar can count and locate the pattern in optimal time using $\mathcal{O}(\gamma \log \frac{n}{\gamma} \log^{\epsilon} n)$ space, where γ is the size of the smallest string attractor of the string [14].

In this paper we extend the results of Ganardi et al. to RLSLPs, that is, we show that one can always balance an RLSLP in linear time without increasing its asymptotic size. This result yields a considerable simplification to the algorithm for accessing any symbol of the string in logarithmic time [5, Appendix A]. It has other implications, like computing range minimum queries (RMQs) [7] or Karp-Rabin fingerprints [12], in $\mathcal{O}(\log n)$ time and within $\mathcal{O}(g_{rl})$ space. We generalize those concepts and show how to compute a wide class of semiring-like functions

over substrings of an RLSLP-compressed string within $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ applications of the function.

2 Terminology

2.1 Strings

Let Σ be any finite set of *symbols* (an *alphabet*). A *string* w is any finite tuple of elements in Σ. The *length* of a string is the length of the tuple, and the *empty string* of length 0 is denoted by ε. The set Σ^* is formed by all the strings that can be defined over Σ. For any string $w = w_1 \dots w_n$, its i-th symbol is denoted by $w[i] = w_i$. Similarly, $w[i:j] = w_i \dots w_j$ with $1 \leq i \leq j \leq n$, or ε if $j < i$. We also define $w[:i] = w_1 \dots w_i$ and $w[i:] = w_i \dots w_n$. If $x[1:n]$ and $y[1:m]$ are strings, the concatenation operation xy is defined as $xy = x_1 \dots x_n y_1 \dots y_m$. If $w = xyz$, then y (resp. x, z) is a *substring* (resp. *prefix*, *suffix*) of w.

2.2 Straight-Line Programs

A *straight-line program* (SLP) is a deterministic context-free grammar generating a unique string w. More formally, an SLP is a context free grammar $G = (V, \Sigma, R, S)$ where V is the set of variables (or non-terminals), Σ is the set of terminal symbols (disjoint from V), $R \subseteq V \times (V \cup \Sigma)^*$ is the set of rules and S is the initial variable; satisfying that each variable has only one rule associated, and that the variables are ordered in such a way that the starting variable is the greater of them, and any variable can only refer to other variables strictly lesser than itself or terminals, in the right-hand side of its rule. Any variable A derives a unique string $\texttt{exp}(A)$, and the string generated by the SLP, is the string generated by its starting variable. The size of an SLP is defined as the sum of the lengths of the right-hand side of its rules. The size of the smallest SLP generating a string is denoted by g, and is a relevant measure of repetitiveness. An SLP generating a non-empty string is often given in so-called Chomsky Normal Form, that is, with all its rules being of the form $A \to BC$ or $A \to a$ for A, B, C variables, and a a terminal symbol.

While computing the smallest grammar is an NP-hard problem [4], there exist several heuristic providing log-approximations of the smallest SLP [11,23]. SLPs are popular as compression devices because several problems over strings can be solved efficiently using their SLP representation, without ever decompressing them. Examples of this are accessing to arbitrary positions of w, extracting substrings, and many other kind of queries [2]. For several queries, it is convenient to have a balanced SLP, that is, an SLP whose parse tree has $\mathcal{O}(\log n)$ depth. Recently, Ganardi et al. showed that any SLP can be balanced [10].

2.3 Directed Acyclic Graph of an SLP

A *directed acyclic graph* (DAG) is a directed multigraph D without cycles (nor loops). We denote by $|D|$ the number of edges in this DAG. For our purposes,

we assume that any DAG has a distinguished node r, satisfying that any other node can be reached from r, and has no incoming edges. We also assume that if a node has k outgoing edges, they are numbered from 1 to k. The *sink nodes* of a DAG are the nodes without outgoing edges. The set of sink nodes of D is denoted by W. We denote the number of paths from u to v as $\pi(u, v)$, and $\pi(u, V) = \sum_{v \in V} \pi(u, v)$ for a set V of nodes. The number of paths from the root to the sink nodes is $n(D) = \pi(r, W)$.

One can interpret an SLP generating a string w as a DAG D: There is a node for each variable in the SLP, the root node is the initial variable, terminal rules of the form $A \to a$ are the sink nodes, and a variable with rule $A \to B_1 B_2 \dots B_k$ has outgoing edges (A, i, B_i) for $i \in [1..k]$. Note that if D is a DAG representing G, then $n(D) = |\mathtt{exp}(G)| = |w|$.

2.4 Run-Length Straight-Line Programs

A *run-length straight-line program* (RLSLP) is an SLP extended with *run-length* rules [21]. An RLSLP can have rules of the form:

- $A \to a$, for some terminal symbol a.
- $A \to A_1 A_2 \dots A_k$, for some variables $A_1, \dots, A_k$ and $k > 1$.
- $A \to B^t$, for some $t > 2$.

The string generated by a variable A with rule $A \to B^t$ is $\mathtt{exp}(B)^t$. A run-length rule is considered to have size 2 (one word is needed to store the exponent). We denote by g_{rl} to the size of the smallest RLSLP generating the string. The depth of the RLSLP is the depth of its associated equivalent SLP, obtained by *unfolding* its run-length rules $A \to B^t$ into rules of the form $A \to BB \dots B$ of length t. Observe that a rule of the form $A \to A_1 A_2 \dots A_k$ can always be transformed into $\mathcal{O}(k)$ rules of size 2, with one of them derivating the same string as A. Doing this for all rules can increase the depth of the RLSLP, but if k is bounded by a constant, then this increase is only by a constant factor.

3 Balancing Run-Length Straight-Line Programs

The idea utilized by Ganardi et al. to transform an SLP G into an equivalent balanced SLP of size $\mathcal{O}(|G|)$ [10, Theorem 1.2], can be adapted to work with RLSLPs. First, we state some definitions and results proved in their work, which we need to obtain our result.

Definition 1. *(Ganardi et al. [10, page 5]) Let D be a DAG, and define the pairs $\lambda(v) = (\lfloor \log_2 \pi(r, v) \rfloor, \lfloor \log_2 \pi(v, W)) \rfloor)$. The* symmetric centroid decomposition (SC-decomposition) *of a DAG D produces a set of edges between nodes with the same λ pairs defined as $E_{scd}(D) = \{(u, i, v) \in E \mid \lambda(u) = \lambda(v)\}$, partitioning D into disjoint paths called* SC-paths *(some of them possibly empty).*

The set E_{scd} can be computed in $\mathcal{O}(|D|)$ time. If D is the DAG of an SLP G this becomes $\mathcal{O}(|G|)$. The following lemma justifies the name "SC-paths".

$A_0 \to A_1A_{12}$
$A_1 \to A_{11}A_2$
$A_2 \to A_5A_3$
$A_3 \to A_4A_6$
$A_4 \to A_5^5$
$A_5 \to A_{11}A_6$
$A_6 \to A_7A_{12}$
$A_7 \to A_8A_{12}$
$A_8 \to A_{10}A_9$
$A_9 \to A_{10}^4$
$A_{10} \to A_{11}A_{12}$
$A_{11} \to \mathtt{0}$
$A_{12} \to \mathtt{1}$

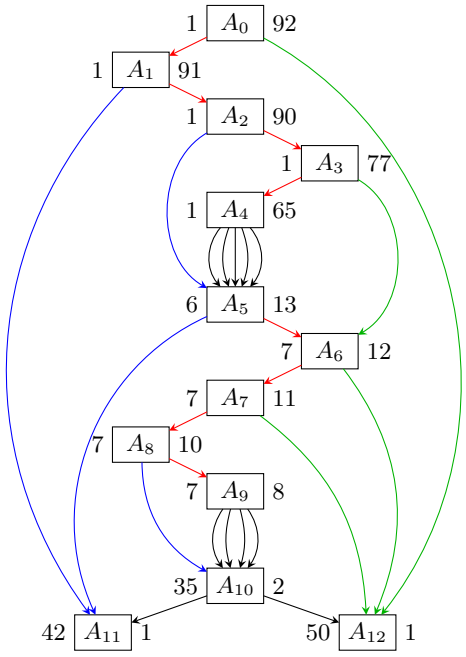

Fig. 1. The DAG and SC-decomposition of an unfolded RLSLP generating the string $\mathtt{0}(\mathtt{0}(\mathtt{01})^5\mathtt{1}^2)^6(\mathtt{01})^5\mathtt{1}^3$. The value to the left of a node is the number of paths from the root to that node, and the value to the right is the number of paths from the node to sink nodes. Red edges belong to the SC-decomposition of the DAG. Blue (resp. green) edges branch from an SC-path to the left (resp. to the right).

Lemma 1. *(Ganardi et al. [10, Lemma 2.1]) Let $D = (V, E)$ be a DAG. Then every node has at most one outgoing and at most one incoming edge from $E_{scd}(D)$. Furthermore, every path from the root r to a sink node contains at most $2\log_2 n(D)$ edges that do not belong to $E_{scd}(D)$.*

Note that the sum of the lengths of all SC-paths is at most the number of nodes of the DAG, or the number of variables of the SLP. An example of the SC-decomposition of a DAG can be seen in Figure 1.

The following definition and technical lemma are needed to construct the building blocks of our balanced RLSLPs.

Definition 2. *(Ganardi et al. [10, page 7])* A weighted string *is a string $w \in \Sigma^*$ equipped with a* weight function *$||\cdot|| : \Sigma \to \mathbb{N}\backslash\{0\}$, which is extended homomorphically. If A is a variable in an SLP G, then we also write $||A||$ for the weight of the string $\mathtt{exp}(A)$ derived from A.*

Lemma 2. *(Ganardi et al. [10, Proposition 2.2]) For every non-empty weighted string w of length n one can construct in linear time an SLP G with the following properties:*

- *G contains at most $3n$ variables*
- *All right-hand sides of G have length at most 4*
- *G contains suffix variables $S_1, ..., S_n$ producing all non-trivial suffixes of w*
- *every path from S_i to some terminal symbol a in the derivation tree of G has length at most $3 + 2(\log_2 ||S_i|| - \log_2 ||a||)$*

We prove that any RLSLP can be balanced without asymptotically increasing its size. Our proof generalizes that of [10, Theorem 1.2] for SLPs.

Theorem 1. *Given an RLSLP G generating a string w, it is possible to construct an equivalent balanced RLSLP G' of size $\mathcal{O}(|G|)$, in linear time, with only rules of the form $A \to a, A \to BC$, and $A \to B^t$, where a is a terminal, B and C are variables, and $t > 2$.*

Proof. Without loss of generality, assume that G has rules of length at most 2, so it is almost in Chomsky Normal Form, except that it has run-length rules. Transform the RLSLP G into an SLP H by unfolding its run-length rules, and then obtain the SC-decomposition $E_{scd}(D)$ of the DAG D of H. Observe that the SC-paths of H use the same variables of G, so it holds that the sum of the lengths of all the SC-paths of H is less than the number of variables of G. Also, note that any variable A of G having a rule of the form $A \to B^t$ for some $t > 2$ is necessarily an endpoint of an SC-path in D, otherwise A would have t outgoing edges in $E_{scd}(D)$, which cannot happen.[1] This implies that the balancing procedure of Ganardi et al. over H, which transforms the rules of variables that are not the endpoint of an SC-path in the DAG D, will not touch variables that originally were run-length in G.

Let $\rho = (A_0, d_0, A_1), (A_1, d_1, A_2), \ldots, (A_{p-1}, d_{p-1}, A_p)$ be an SC-path of D. It holds that for each A_i with $i \in [0..p-1]$, in the SLP H, its rule goes to two distinct variables, one to the left and one to the right. For each variable A_i, with $i \in [0..p-1]$, there is a variable A'_{i+1} that is not part of the path. Let $A'_1 A'_2 \ldots A'_p$ be the sequence of those variables. Let $L = L_1 L_2 \ldots L_s$ be the subsequence of left variables of the previous sequence. Then construct an SLP of size $\mathcal{O}(s) \leq \mathcal{O}(p)$ for the sequence L (seen as a string) as in Lemma 2, using $|\texttt{exp}(L_i)|$ in H as the weight function. In this SLP, any path from the suffix nonterminal S_i to a variable L_j has length at most $3 + 2(\log_2 ||S_i|| - \log_2 ||L_j||)$. Similarly, construct an SLP of size $\mathcal{O}(t) \leq \mathcal{O}(p)$ for the sequence $R = R_1 R_2 \ldots R_t$ of right symbols in reverse order, as in Lemma 2, but with prefix variables P_i instead of suffix variables. Each variable A_i, with $i \in [0..p-1]$, derives the same string as $w_\ell A_p w_r$, for some suffix w_ℓ of L and some prefix w_r of R. We can find rules deriving these prefixes and suffixes in the SLPs produced in the previous step, so for any variable A_i, we construct an equivalent rule of length at most 3. Add these equivalent rules, and the left and right SLP rules to a new RLSLP G'. Do this for all SC-paths. Finally, we add the original terminal variables and run-length variables of the RLSLP G, so G' is an RLSLP equivalent to G.

[1] Seen another way, $\lambda(A) \neq \lambda(B)$ because $\log_2 \pi(A, W) = \log_2(t \cdot \pi(B, W)) > 1 + \log_2 \pi(B, W)$.

The SLP constructed for L has all its rules of length at most 4, and $3s \leq 3p$ variables. The same happens with R. The other constructed rules also have length at most 3, and there are p of them. Summing over all SC-paths we have $\mathcal{O}(|G|)$ size. The original terminal variables and run-length variables of G have rules of size at most 4, and we keep them. Thus, the RLSLP G' has size $\mathcal{O}(|G|)$.

Any path in the derivation tree of G' is of length $\mathcal{O}(\log n)$. To see why, let $A_0, \ldots, A_p$ be an SC-path. Consider a path from a variable A_i to an occurrence of a variable that is in the right-hand side of A_p in G'. Clearly this path has length at most 2. Now consider a path from A_i to a variable A'_j in L with $i < j \leq p$. By construction this path is of the form $A_i \to S_k \to^* A'_j$ for some suffix variable S_k (if the occurrence of A'_j is a left symbol), and its length is at most $1 + 3 + 2(\log_2 ||S_k|| - \log_2 ||A'_j||) \leq 4 + 2\log_2 ||A_i|| - 2\log_2 ||A'_j||$. Analogously, if A'_j is a right variable, the length of the path is bounded by $1+3+2(\log_2 ||P_k|| - \log_2 ||A'_j||) \leq 4+2\log_2 ||A_i|| - 2\log_2 ||A'_j||$. Finally, consider a maximal path to a leaf in the parse tree of G'. Factorize it as

$$A_0 \to^* A_1 \to^* \cdots \to^* A_k$$

where each A_i is a variable of H (and also of G). Paths $A_i \to^* A_{i+1}$ are like those defined in the paragraph above, satisfying that their length is bounded by $4 + 2\log_2 ||A_i|| - 2\log_2 ||A_{i+1}||$. Observe that between each A_i and A_{i+1}, in the DAG D there is almost an SC-path, except that the last edge is not in E_{scd}. The length of this path is at most

$$\sum_{i=0}^{k-1}(4 + 2\log_2 ||A_i|| - 2\log_2 ||A_{i+1}||) \leq k + 2\log_2 ||A_0|| - 2\log_2 ||A_k||$$

By Lemma 1, $k \leq 2\log_2 n$, which yields the $\mathcal{O}(\log n)$ upper bound. The construction time is linear, because the SLPs of Lemma 2 are constructed in linear time in the lengths of the SC-paths (summing to $\mathcal{O}(|G|)$), and $E_{scd}(D)$ can be obtained in time $\mathcal{O}(|G|)$ (instead of $\mathcal{O}(H)$) if we represent in the DAG D the edges of a variable A with rule $A \to B^t$ as a single edge extended with the power t. This way, when traversing the DAG from root to sinks and sinks to root to compute λ values, it holds that $\pi(A, W) = t \cdot \pi(B, W)$, and that $\pi(r, B) = t \cdot \pi(r, A) + c$, where c are the paths from root incoming from other variables. Thus, each run-length edge must be traversed only once, not t times.

To have rules of size at most two, delete rules in G' of the form $A \to B$ (replacing all A's by B's), and note that rules of the form $A \to BCDE$ or $A \to BCD$ can be decomposed into rules of length 2, with only a constant increase in size and depth. □

4 Substring Range Operations in $\mathcal{O}(g_{rl})$ space

4.1 Karp-Rabin Fingerprints

To answer signature $\kappa(w[p:q]) = (\sum_{i=p}^{q} w[i] \cdot c^{i-p}) \bmod \mu$, for a suitable integer c and prime number μ, we use the following identity for any $p' \in [p..q-1]$:

$$\kappa(w[p:q]) = \left(\kappa(w[p:p']) + \kappa(w[p'+1:q]) \cdot c^{p'-p+1}\right) \bmod \mu \tag{1}$$

and then it holds

$$\kappa(w[p:p']) = \left(\kappa(w[p:q]) - \kappa(w[p'+1:q]) \cdot c^{p'-p+1}\right) \bmod \mu$$
$$\kappa(w[p'+1:q])\cdot = \left(\frac{\kappa(w[p:q]) - \kappa(w[p:p'])}{c^{p'-p+1}}\right) \bmod \mu,$$

which implies that, to answer $\kappa(w[p:q])$, we can compute $\kappa(w[1:p-1])$ and $\kappa(w[1:q])$ and then subtract one to another. For that reason, we only consider computing fingerprints of text prefixes. Then, the recursive calls to our algorithm just need to know the right boundary of a prefix, namely computing signature on the substring $\texttt{exp}(A)[1:j]$ of the string expanded by a symbol A of our grammar can be expressed as $\kappa(A, j)$.

Suppose that we want to compute the signature of a prefix $w[1:j]$ and that there is a rule $A \to BC$ such that $\texttt{exp}(A) = w[1:q]$, with $j \leq q$. If $j = |\texttt{exp}(B)|$ or $j = q$, we can have stored $\kappa(\texttt{exp}(A))$ and $\kappa(\texttt{exp}(B))$ and answer directly the query. On the other hand, if $j < |\texttt{exp}(B)|$, we can answer $\kappa(\texttt{exp}(B)[1:j])$ by descending in the derivation tree of B. Otherwise, $|\texttt{exp}(B)| < j < |\texttt{exp}(A)|$, then we can use Eq. 1 and answer $(\kappa(\texttt{exp}(B)) + \kappa(\texttt{exp}(C)[1 : j - |\texttt{exp}(B)|] \cdot c^{|\texttt{exp}(B)|}) \bmod \mu$, where $\kappa(\texttt{exp}(C)[1 : j - |\texttt{exp}(B)|]$ is obtained by descending in the derivation tree of C. Then, in addition to storing $\kappa(\texttt{exp}(A))$ for every nonterminal A, we also need to store $c^{|\texttt{exp}(A)|} \bmod \mu$ and $|\texttt{exp}(A)|$. Therefore, the cost of computing fingerprints is just the depth of the derivation tree of A.

The same does not apply for run-length rules $A \to B^t$, because we cannot afford the space consumption of storing $c^{t' \cdot |\texttt{exp}(B)|} \bmod \mu$ for every $1 \leq t' \leq t$, as this could give us a structure bigger than $\mathcal{O}(g_{rl})$. Instead, we can treat run-length rules as regular rules $A \to B \dots B$. Then, we can use the following identity

$$\kappa(\texttt{exp}(B^{t'})) = \left(\kappa(\texttt{exp}(B)) \cdot \frac{c^{|\texttt{exp}(B)| \cdot t'} - 1}{c^{|\texttt{exp}(B)|} - 1}\right) \bmod \mu.$$

Namely, to compute $\kappa(\texttt{exp}(B^{t'}))$ we can have previously stored $c^{|\texttt{exp}(B)|} \bmod \mu$ and $(c^{|\texttt{exp}(B)|} - 1)^{-1} \bmod \mu$ and then compute the exponentiation in time $\mathcal{O}(\log t)$. With this, if $j \in [t' \cdot |\texttt{exp}(B)| + 1..(t'+1) \cdot |\texttt{exp}(B)|]$ we can handle run-length rules signatures $\kappa(\texttt{exp}(B^t)[1:j])$ as

$$\left(\kappa(\texttt{exp}(B^{t'})) + \kappa(\texttt{exp}(B)[1 : j - t' \cdot |\texttt{exp}(B)|]) \cdot c^{t' \cdot |\texttt{exp}(B)|}\right) \bmod \mu,$$

where $\kappa(\texttt{exp}(B)[1 : j - t' \cdot |\texttt{exp}(B)|])$ is obtained by descending in the derivation tree of B. We are saving space by storing our structure at the cost of increasing computation time. As we show later, this time is in fact logarithmic.

A structure for Karp-Rabin signatures. We construct a structure over a balanced RLSLP from Theorem 1, using some auxiliary arrays. We define an array $L[A] = |\mathtt{exp}(A)|$ consisting of the length of the expansion of each nonterminal A. For terminals a, we assume $L[a] = 1$. Also, we define arrays K_1 and K_2 such that, for each nonterminal A,

$$K_1[A] = \kappa(\mathtt{exp}(A)),$$
$$K_2[A] = c^{L[A]} \bmod \mu,$$

with the Karp-Rabin fingerprint of the string expanded by A and the last power of c used in the signature multiplied by c, namely the first power needed for signing the second part of the string expanded by A. For terminals a we assume $K_1[a] = a \bmod \mu$ and $K_2[a] = c \bmod \mu$. In addition, for rules $A \to B^t$ we store

$$E[A] = (K_2[B] - 1)^{-1} \bmod \mu.$$

The arrays L, K_j, and E add only $\mathcal{O}(g_{rl})$ extra space. With these auxiliary structures, we can compute fingerprints in $\mathcal{O}(\log n)$ time.

Theorem 2 (cf. [1,5]). *It is possible to construct an index of size $\mathcal{O}(g_{rl})$ supporting Karp-Rabin fingerprints for prefixes of $w[1:n]$ in $\mathcal{O}(\log n)$ time.*

Proof. Let G be a balanced RLSLP of size $\mathcal{O}(g_{rl})$ constructed as in Theorem 1. We construct arrays L, K_i, and E as shown above. To compute $\kappa(A, j)$, we do as follows:

1. If $j = L[A]$, return $K_1[A]$.
2. If $A \to BC$, then:
   (a) If $j \le L[B]$, return $\kappa(B, j)$.
   (b) If $L[B] < j$, return $\big(K_1[B] + \kappa(C, j - L[B]) \cdot K_2[B]\big) \bmod \mu$.
3. If $A \to B^t$ for $t > 2$, then:
   (a) If $j \le L[B]$, return $\kappa(B, j)$.
   (b) If $j \in [t'L[B] + 1..(t' + 1)L[B]]$ with $1 \le t' < t$, let $e = K_2[B]^{t'}$ and $f = (e - 1) \cdot E[A] \bmod \mu$, then return

$$\big(K_1[B] \cdot f + \kappa(B, j - t'L[B]) \cdot e\big) \bmod \mu.$$

Every step of the algorithm takes $\mathcal{O}(1)$ time, so the cost is the depth of the derivation tree of G. The only exception is case 3(b), in which we have an exponentiation. For a non-terminal $A \to B^t$, this exponentiation takes $\mathcal{O}(\log t)$ time, which is $\mathcal{O}(\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|))$ time for managing every run-length rule. We show next that $\mathcal{O}(\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|))$ telescopes to $\mathcal{O}(\log |\mathtt{exp}(A)|)$, thus we obtain $\mathcal{O}(\log n)$ time for the overall algorithm time.

The telescoping argument is as follows. We prove by induction that the cost $k(A)$ to compute $\kappa(A, j)$ is at most $h(A) + \log |\mathtt{exp}(A)|$, where $h(A)$ is the height of the parse tree of A and j is arbitrary. Then in case 2 we have $k(A) \le 1 + \mathtt{max}(k(B), k(C))$, which by induction is $\le 1 + \mathtt{max}(h(B), h(C)) + \log |\mathtt{exp}(A)| = h(A) + \log |\mathtt{exp}(A)|$. In case 3 we have $k(A) \le 1 + \log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|) + k(B)$, and since by induction $k(B) \le h(B) + \log |\mathtt{exp}(B)|$, we obtain $k(A) \le h(A) + \log |\mathtt{exp}(A)|$. Since G is balanced, this implies $k(A) = \mathcal{O}(\log n)$ when A is the root symbol. □

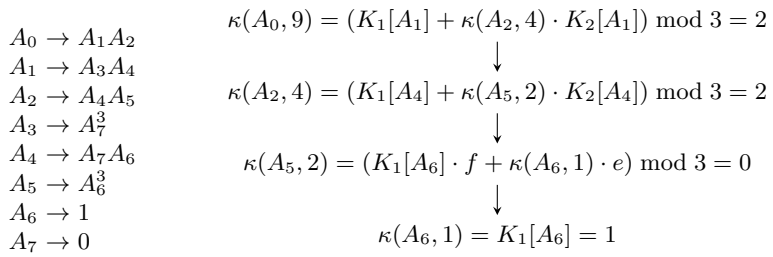

Fig. 2. Example of a balanced RLSLP for the string 0^4101^4 (left) and fingerprint computation over a length-8 prefix of the string generated by this RLSLP (right), with $c = 2$, $\mu = 3$, $K_1[A_1, A_4, A_6] = [1, 2, 1]$, $K_2[A_1, A_4, A_6] = [2, 1, 2]$, $f = 1$, and $e = 2$.

Figure 2 shows an example of this procedure.

4.2 Range Minimum Queries

A *range minimum query* (RMQ) over a string returns the position of the leftmost occurrence of the minimum within a range. For these type of queries, we can provide an $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ time solution. In the Appendix A we also show how to efficiently compute the related PSV/NSV queries.

Theorem 3. *It is possible to construct an index of size $\mathcal{O}(g_{rl})$ supporting RMQs in $\mathcal{O}(\log n)$ time.*

Proof. Let G be a balanced RLSLP of size $\mathcal{O}(g_{rl})$ constructed as in Theorem 1. We define $\texttt{rmq}(A, i, j)$ as the pair (a, k) where a is the least symbol in $\texttt{exp}(A)[i : j]$, and k is the absolute position within $\texttt{exp}(A)$ of the leftmost occurrence of a in $\texttt{exp}(A)[i : j]$. Store the values $L[A] = |\texttt{exp}(A)|$, and $M[A] = \texttt{rmq}(A, 1, L[A])$, for every variable A, as arrays. These arrays add only $\mathcal{O}(g_{rl})$ extra space. To compute $\texttt{rmq}(A, i, j)$, do as follows:

1. If $i = 1$ and $j = L[A]$, return $M[A]$.
2. If $A \to BC$, then:
   (a) If $i, j \leq L[B]$, return $\texttt{rmq}(B, i, j)$.
   (b) If $i, j > L[B]$, let $(a, k) = \texttt{rmq}(C, i-L[B], j-L[B])$. Return $(a, L[B]+k)$.
   (c) If $i \leq L[B]$ and $L[B] < j$ with $j-i+1 < L[A]$, let $(a_1, k_1) = \texttt{rmq}(B, i, L[B])$ and $(a_2, k_2) = \texttt{rmq}(C, 1, j-L[B])$. Return (a_1, k_1) if $a_1 \leq a_2$, or $(a_2, L[B]+k_2)$ if $a_2 < a_1$.
3. If $A \to B^t$ for $t > 2$, then:
   (a) If $i, j \in [t'L[B]+1..(t'+1)L[B]]$, let $(a, k) = \texttt{rmq}(B, i-t'L[B], j-t'L[B])$. Return $(a, t'L[B] + k)$
   (b) If $i \in [t'L[B]+1..(t'+1)L[B]]$ and $j \in [t''L[B]+1..(t''+1)L[B]]$ for some $t' < t''$. Let $(a_l, k_l) = \texttt{rmq}(B, i - t'L[B], L[B])$, $(a_r, k_r) = \texttt{rmq}(B, 1, j - t''L[B])$ and $(a_c, k_c) = M[B]$ (only if $t''-t' > 1$). Return (a, k), where $a = \texttt{min}(a_l, a_r, a_c)$, and k is either $t'L[B]+k_l$, $t''L[B]+k_r$, or $(t'+1)L[B]+k_c$ (only if $t'' - t' > 1$), depending on which of these positions correspond to an absolute position of a in $\texttt{exp}(A)$, and is the leftmost of them.

We analyze the number of recursive calls of the algorithm above. For cases 2(a), 2(b) and 3(a) there is only one recursive call, over a variable which is deeper in the derivation tree of G. In cases 2(c) and 3(b), it could be that two recursive calls occur, but overall, this can happen only one time in the whole run of the algorithm. The reason is that when two recursive calls occur at the same depth, from that point onward, the algorithm will be computing $\texttt{rmq}(\cdot)$ over suffixes or prefixes of expansions of variables. If we try to compute for example $\texttt{rmq}(A, i, L[A])$, and A is of the form $A \to BC$, if $i < L[B]$, the call over B is again a suffix call. If $A \to B^t$ for some $t > 2$, and we want to compute $\texttt{rmq}(A, i, L[A])$, we end with a recursive call over a suffix of B too. Hence, there are only $\mathcal{O}(\log n)$ recursive calls to $\texttt{rmq}(\cdot)$. The non-recursive step takes constant time, even for run-length rules, so we obtain $\mathcal{O}(\log n)$ time. □

4.3 More general functions

More generally, we can compute a wide class of functions in $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ applications of the function.

Theorem 4. *Let f be a function from strings to a set of size $n^{\mathcal{O}(1)}$, such that $f(xy) = h(f(x), f(y), |x|, |y|)$ for any strings x and y, where h is a function computable in time $\mathcal{O}(\texttt{time}(h))$. Let $w[1:n]$ be a string. It is possible to construct an index to compute $f(w[i:j])$ in $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\texttt{time}(h) \cdot \log n)$ time.*

Proof. Let G be a balanced RLSLP of size g_{rl} constructed as in Theorem 1. Store the values $L[A] = |\texttt{exp}(A)|$ and $F[A] = f(\texttt{exp}(A))$, for every variable A, as arrays. These arrays add only $\mathcal{O}(g_{rl})$ extra space because the values in F fit in $\mathcal{O}(\log n)$-bit words. To compute $f(A, i, j) = f(\texttt{exp}(A)[i:j])$, we do as follows:

1. If $i = 1$ and $j = L[A]$, return $F[A]$.
2. If $A \to BC$, then:
   (a) If $i, j \leq L[B]$, return $f(B, i, j)$.
   (b) If $i, j > L[B]$, return $f(C, i - L[B], j - L[B])$.
   (c) If $i \leq L[B] < j$, return
   $$h(f(B, i, L[B]), f(C, 1, j - L[B]), L[B] - i + 1, j - L[B]).$$
3. If $A \to B^t$ for $t > 2$, then:
   (a) If $i, j \in [t'L[B] + 1..(t' + 1)L[B]]$, return $f(B, i - t'L[B], j - t'L[B])$.
   (b) If $i \in [t'L[B]+1..(t'+1)L[B]]$ and $j \in [t''L[B]+1..(t''+1)L[B]]$ for some $t' < t''$, let
   $$\begin{aligned} f_l &= f(B, i - t'L[B], L[B]) \\ f_r &= f(B, 1, j - t''L[B]) \\ f_c(0) &= f(\varepsilon) \\ f_c(1) &= F[B] \\ f_c(i) &= h(f_c(i/2), f_c(i/2), L[B]^{i/2}, L[B]^{i/2}) \text{ for even } i \\ f_c(i) &= h(f_c(1), f_c(i-1), L[B], L[B]^{i-1}) \text{ for odd } i \\ h_l &= h(f_l, f_c(t'' - t' - 1), (t' + 1)L[B] - i + 1, (t'' - t' - 1)L[B]) \end{aligned}$$

then return

$$h(h_l, f_r, (t'+1)L[B] - i + 1 + (t'' - t' - 1)L[B], j - t''L[B] + 1)$$

Just like when computing RMQs in Theorem 3, there is at most one call in the whole algorithm invoking two non-trivial recursive calls. To estimate the cost of each recursive call, the same analysis as for Theorem 2 works, because the expansion of whole nonterminals is handled in constant time as well, and the $\mathcal{O}(\log t)$ cost of the run-length rules telescopes in the same way.

The precise telescoping argument is as follows. We prove by induction that the cost $c(A)$ to compute $f(A, i, L[A])$ or $f(A, 1, j)$ (i.e., the cost of suffix or prefix calls) is at most $\texttt{time}(h) \cdot (h(A) + \log|\texttt{exp}(A)|)$, where $h(A)$ is the height of the parse tree of A and i, j are arbitrary. Then in case 2 we have $c(A) \leq \texttt{time}(h) + \texttt{max}(c(B), c(C))$, which by induction is at most $\texttt{time}(h) \cdot (1 + \texttt{max}(h(B), h(C)) + \log|\texttt{exp}(A)|) = \texttt{time}(h) \cdot (h(A) + \log|\texttt{exp}(A)|)$. In case 3 we have that the cost is $c(A) \leq \texttt{time}(h) \cdot \log(|\texttt{exp}(A)|/|\texttt{exp}(B)|) + c(B)$, which by induction yields

$$\begin{aligned} c(A) &\leq \texttt{time}(h) \cdot (\log(|\texttt{exp}(A)|/|\texttt{exp}(B)|) + h(B) + \log|\texttt{exp}(B)|)) \\ &\leq \texttt{time}(h) \cdot (h(A) + \log|\texttt{exp}(A)|) \end{aligned}$$

In the case that two non-trivial recursive calls are made at some point when computing $f(A_k, i, j)$, this is the unique point in the algorithm where it happens, so we charge only $\texttt{time}(h) \cdot (h(A_k) + \log|\texttt{exp}(A_k)|)$ to the cost of A_k. Then the total cost of the algorithm starting from A_0 is at most $\texttt{time}(h) \cdot (h(A_0) + \log|\texttt{exp}(A_0)|)$ plus the cost $\texttt{time}(h) \cdot (h(A_k) + \log|\texttt{exp}(A_k)|)$ that we did not charge to A_k. This at most doubles the cost, maintaining it within the same order. Because the grammar is balanced, we obtain $\mathcal{O}(\texttt{time}(h) \cdot \log n)$ time. □

5 Conclusion

In this work, we have shown that any RLSLP can be balanced in linear time without increasing it asymptotic size. This allows us to compute several substring range queries like RMQ, PSV/NSV (in the Appendix A), and Karp-Rabin fingerprints $\mathcal{O}(\log n)$ time within $\mathcal{O}(g_{rl})$ space. More generally, in $\mathcal{O}(g_{rl})$ space we can compute the wide class of substring functions that satisfy $f(xy) = h(f(x), f(y), |x|, |y|)$, in $\mathcal{O}(\log n)$ times the cost of computing h. Our work also simplifies some previously established results like retrieving substrings in $\mathcal{O}(\log n)$ space and within $\mathcal{O}(g_{rl})$ space.

An open challenge is to efficiently count the number of occurrences of a pattern in the string, within $\mathcal{O}(g_{rl})$ space [5, Appendix A].

References

1. Bille, P., Gørtz, I.L., Cording, P.H., Sach, B., Vildhøj, H.W., Vind, S.: Fingerprints in compressed strings. Journal of Computer and System Sciences **86**, 171–180 (2017). https://doi.org/https://doi.org/10.1016/j.jcss.2017.01.002, https://www.sciencedirect.com/science/article/pii/S0022000017300028

2. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM Journal on Computing **44**(3), 513–539 (2015), https://doi.org/10.1137/130936889
3. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Tech. rep., DIGITAL SRC RESEARCH REPORT (1994)
4. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory **51**(7), 2554–2576 (2005)
5. Christiansen, A., Ettienne, M., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. ACM Transactions on Algorithms **17**, 1–39 (12 2020). https://doi.org/10.1145/3426473
6. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science **410**(51), 5354–5364 (2009)
7. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays **40**(2) (2011), https://doi.org/10.1137/090779759
8. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in bwt-runs bounded space. In: Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1459–1477 (2018)
9. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. J. ACM **67**(1) (jan 2020). https://doi.org/10.1145/3375890, https://doi.org/10.1145/3375890
10. Ganardi, M., Jeż, A., Lohrey, M.: Balancing straight-line programs. J. ACM **68**(4) (jun 2021), https://doi.org/10.1145/3457389
11. Jeż, A.: Approximation of grammar-based compression via recompression. Theoretical Computer Science **592**, 115–134 (2015)
12. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (mar 1987), https://doi.org/10.1147/rd.312.0249
13. Kempa, D., Kociumaka, T.: Resolution of the burrows-wheeler transform conjecture. Commun. ACM **65**(6), 91–98 (may 2022), https://doi.org/10.1145/3531445
14. Kempa, D., Prezza, N.: At the roots of dictionary compression: string attractors. Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (Jun 2018). https://doi.org/10.1145/3188745.3188814, http://dx.doi.org/10.1145/3188745.3188814
15. Kini, D., Mathur, U., Viswanathan, M.: Data race detection on compressed traces. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 26–37. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236025, https://doi.org/10.1145/3236024.3236025
16. Kreft, S., Navarro, G.: Lz77-like compression with fast random access. In: 2010 Data Compression Conference. pp. 239–248 (2010)
17. Larsson, N., Moffat, A.: Offline dictionary-based compression. In: Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096). pp. 296–305 (1999)
18. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Trans. Inf. Theory **22**(1), 75–81 (1976)
19. Navarro, G.: Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Computing Surveys **54**(2), article 29 (2021)
20. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: A linear-time algorithm **7**(1), 67–82 (sep 1997)

21. Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Fully Dynamic Data Structure for LCE Queries in Compressed Space. In: 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 58, pp. 72:1–72:15 (2016)
22. Przeworski, M., Hudson, R., Di Rienzo, A.: Adjusting the focus on human variation. Trends in genetics : TIG **16**(7), 296—302 (July 2000)
23. Rytter, W.: Application of lempel–ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science **302**(1), 211–222 (2003)
24. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 247–258 (2013)
25. Zhang, M., Mathur, U., Viswanathan, M.: Checking ltl[f,g,x] on compressed traces in polynomial time. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 131–143. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3468557, https://doi.org/10.1145/3468264.3468557

A PSV and NSV queries

Other relevant queries are *previous smaller value* (PSV) and *next smaller value* (NSV) [6,9], defined as follows:

- $\texttt{psv}(i) = \texttt{max}(\{j \mid j < i, w[j] < w[i]\} \cup \{0\})$
- $\texttt{nsv}(i) = \texttt{min}(\{j \mid j > i, w[j] < w[i]\} \cup \{n+1\})$
- $\texttt{psv}'(i,d) = \texttt{max}(\{j \mid j < i, w[j] < d\} \cup \{0\})$
- $\texttt{nsv}'(i,d) = \texttt{min}(\{j \mid j > i, w[j] < d\} \cup \{n+1\})$

Note that the first two queries can be computed by accessing $w[i]$ in $\mathcal{O}(\log n)$ time, and then calling one of the latter two queries, respectively. We show that the latter queries can be answered in $\mathcal{O}(g_{rl})$ space and $\mathcal{O}(\log n)$ time.

Theorem 5. *It is possible to construct an index of size $\mathcal{O}(g_{rl})$ supporting PSV and NSV queries in $\mathcal{O}(\log n)$ time.*

Proof. Let G be a balanced RLSLP of size $\mathcal{O}(g_{rl})$ constructed as in Theorem 1. Store the values $L[A] = |\texttt{exp}(A)|$ and $M[A] = \texttt{min}(\{\texttt{exp}(A)[i] \mid i \in [1..L[A]]\})$, for every variable A, as arrays. These arrays add only $\mathcal{O}(g_{rl})$ extra space. To compute $\texttt{psv}'(A,i,d)$, do as follows:

1. If $i = 1$ or $M[A] \geq d$, return 0.
2. If $A \to a$, return 1.
3. If $A \to BC$, then:
   (a) If $i \leq L[B] + 1$, return $\texttt{psv}'(B,i,d)$.
   (b) If $L[B] + 1 < i$, let $k = \texttt{psv}'(C, i - L[B], d)$. If $k > 0$, return $L[B] + k$, otherwise, return $\texttt{psv}'(B,i,d)$.
4. If $A \to B^t$ for $t > 2$, then:

(a) If $i \leq L[B] + 1$, return $\mathtt{psv}'(B, i, d)$.
(b) If $i \in [t'L[B] + 1..(t' + 1)L[B]]$, let $k = \mathtt{psv}'(B, i - t'L[B], d)$. If $k > 0$, return $t'L[B] + k$. Otherwise, return $(t' - 1)L[B] + \mathtt{psv}'(B, i, d)$.
(c) If $L[A] < i$, return $(t - 1)L[B] + \mathtt{psv}'(B, i, d)$.

The guard in point 1 guarantees that, in the simple case where i is beyond $|\mathtt{exp}(A)|$, at most one recursive call needs more than $\mathcal{O}(1)$ time. In general, we can make two calls in case 3(b), but then the second call (inside B) is of the simple type from there on. The case of run-length rules is similar. Thus, we obtain $\mathcal{O}(\log n)$ time. The query $\mathtt{nsv}'$ is handled similarly. □