Space-efficient conversions from SLPs *

Travis Gagie[1,5], Adrián Goga[2], Artur Jeż[3], Gonzalo Navarro[4,5]

[1] Faculty of Computer Science, Dalhousie University, Canada
[2] Department of Computer Science, Comenius University in Bratislava, Slovakia
[3] Institute of Computer Science, University of Wrocław, Poland
[4] Department of Computer Science, University of Chile, Chile
[5] CeBiB — Center for Biotechnology and Bioengineering, Chile

Abstract. We give algorithms that, given a straight-line program (SLP) with g rules that generates (only) a text $T[1..n]$, build within $O(g)$ space the Lempel-Ziv (LZ) parse of T (of z phrases) in time $O(n \log^2 n)$ or in time $O(gz \log^2(n/z))$. We also show how to build a locally consistent grammar (LCG) of optimal size $g_{lc} = O(\delta \log \frac{n}{\delta})$ from the SLP within $O(g + g_{lc})$ space and in $O(n \log g)$ time, where δ is the substring complexity measure of T. Finally, we show how to build the LZ parse of T from such an LCG within $O(g_{lc})$ space and in time $O(z \log^2 n \log^2(n/z))$. All our results hold with high probability.

1 Introduction

With the rise of enormous and highly repetitive text collections [32], it is becoming practical, and even necessary, to maintain the collections compressed all the time. This requires being able to perform all the needed computations, like text searching and mining, directly on the compressed data, without ever decompressing it.

As an example, consider the modest (for today's standards) genomic repository *1000 Genomes* [12] containing the genomes of 2,500 individuals. At the typical rate of about 3 billion bases each, the collection would occupy about 7 terabytes. Recent projects like the *Million Genome Initiative*[6] would then require petabytes. The 1000 Genomes project stores and distributes its data already in a compressed form[7] to exploit the fact that, compared to a reference genome, each individual genome has only one difference every roughly 500 bases, on average.

* Research supported by the European Union's Horizon 2020 research and innovation program under Marie Skłodowska-Curie grant agreement No 956229 (ALPACA) and by grants 1/0463/20 and 1/0538/22 from the Scientific Grant Agency of the Ministry of Education, Science, Research, and Sport of the Slovak Republic and Slovak Academy of Sciences (VEGA) and grant APVV-22-0143 from the Slovak Research and Development Agency. T.G. and G.N. funded in part by Basal Funds FB0001, ANID, Chile. T.G. funded in part by NSERC RGPIN-07185-2020. G.N. funded in part by Fondecyt Grant 1-230755, ANID, Chile.

[6] `https://digital-strategy.ec.europa.eu/en/policies/1-million-genomes`

[7] In VCF, `https://github.com/samtools/hts-specs/blob/master/VCFv4.3.pdf`

Certainly one would like to manipulate even such a modest collection always in a compressed form, using gigabytes instead of terabytes of memory!

Some compression formats are more useful for some tasks than others, however. For example, Lempel-Ziv compression [29] tends to achieve the best compression ratios, which makes it more useful for storage and transmission. Grammar compression [26] yields slightly larger files, but in exchange it can produce T in streaming form, and provide direct access to any text snippet [7], as well as indexed searches [11]. Locally consistent grammars provide faster searches, and support more complex queries, while still being bounded by well-known repetitiveness measures [10, 28, 27, 23, 16, 33]. The run-length-encoded Burrows-Wheeler Transform of T requires even more space [22], but in exchange it enables full suffix tree functionality [13].

It is of interest, then, to *convert* from one format to another. Doing this conversion by decompressing the current format and then compressing to the new one is impractical, as it is bound to use $\Omega(n)$ space, which in practice implies running $\Theta(n)$-time algorithms on secondary storage. Thus the interest in algorithms whose running time and space usage can be bounded in terms of input and output size. We say that a conversion between different compression formats is a *fully compressed conversion* when it uses space and time polynomial in the size of the (compressed) input, the size of the (compressed) output, and $\log n$; it is a *compressed conversion* when the bound applies only to space (and so the running time may polynomially depend on n). There is a long line of research on compressed conversions, we recall it below. For brevity we omit a large body of work on producing compressed representations from the original string S, aiming to use little space on top of S itself, and the work on compression formats that are too weak for repetitive data, like LZ78 or run-length compression of the text.

Let z, g, g_{lc} and r be the asymptotic (i.e., up to constant factors) sizes of the Lempel-Ziv (LZ) parse of a string $T[1..n]$, a straight-line program (SLP) or context-free grammar that expands to T, a locally consistent grammar (LCG) that expands to T, and the run-length encoded Burrows-Wheeler Transform (RLBWT) for T, respectively. On highly repetitive texts, all the given measures can be exponentially smaller than n, hence the relevance of such conversions. We refer to some SLP because finding the smallest SLP generating a given string is NP-complete [8]. It holds that $z \leq g \leq g_{lc} \leq r$ in practice. The first such conversion was implicitly given by Mehlhorn, Sundar and Uhrig [30], who proposed a data structure for a dynamic collection of strings allowing adding concatenations and substrings of strings in the collection in polylogarithmic time. The data structure implicitly used (a variant of) a LCG and so it allowed compressed conversions from SLP and LZ to LCG, in time $O(g \log n (\log g \log^* n + \log n))$ and space $O(g \log n \log^* n)$ and time $O(z \log n (\log g \log^* n + \log n))$ and space $O(z \log n \log^* n)$, respectively. They also proposed a randomised variant of the data structure, with which the conversion had expected time and space $O(g \log^2 n)$, $O(g \log n)$ and $O(z \log^2 n)$, $O(z \log n)$, respectively. Their data structure was improved by Alstrup, Brodal and Rauhe [1], who mainly added new functionalities and improved the conversion times to $O(g \log n \log^* n)$ and

$O(z \log n \log^* n)$ w.h.p. (the space usage remained the same). Rytter [38] studied the problem of constructing the smallest SLP for a given string and showed how to build an SLP of size $g = O(z \log(n/z))$ within $O(g)$ space and time from the LZ parse of T in the non-overlapping case (i.e., when phrases cannot overlap their sources), and Gawrychowski [15, Lemma 8] extended this result to the general LZ parse. Nishimoto et al. [36] gave an algorithm constructing the LZ parse from the LCG of Mehlhorn et al. [30], with running time $O(z \log g_{lc} \log^3 n (\log^* n)^2)$ and linear-space. It can also be used to convert an SLP to the LZ parse in time $O(n(\log \log n)^2 + z \log^4 n (\log^* n)^2)$ or $O(n\sqrt{\log z + \log \log n} + z \log^4 n (\log^* n)^2)$ and $O(z \log n \log^* n)$ space. Tomohiro I [18] proposed a conversion algorithm from an SLP to (a variant of) a LCG using $O(g \log(n/g))$ time and $O(g + z \log(n/z))$ space; one can also transform an LZ77 to SLP using with $\log(n/z)$ blowup and then apply the reduction to LCG, using $O(z \log^2(n/z))$ time and $O(z \log(n/z))$ space. Kempa and Kociumaka [23] built on the produced LCG, showing how to convert a LCG or a SLP to the LZ parse in time $O(g_{lc} \log^4 n)$ or $O(g \log^4 n)$, respectively. They also gave a fully compressed conversion from a SLP or the LZ parse to a LCG (of optimal size $O(\delta \log n/\delta))$ in time $O(\delta \log^7 n)$ (δ is another compression measure with $\delta \leq z \leq \delta \log n$ [28]). Policriti and Prezza [37] showed how to convert from the RLBWT to the LZ parse in $O(r + z)$ space and $O(n \log r)$ time, and back in the same space and $O(n \log(rz))$ time. The earlier mentioned paper of Kempa and Kociumaka [22] also converts from the LZ parse to the RLBWT in $O(z \log^8 n)$ expected time. Arimira et al. [2] recently showed how to convert from the compressed directed acyclic word graph (CDAWG) of size e to either RLBWT or LZ, both in $O(e)$ time and space, though e is the weakest among the commonly accepted repetitiveness measures [32].

Note that our contribution deals only with LZ, SLP and LCG; we recalled results for other compression formats (RLBWT, CDAWG) for comparison and to present the state of the art in the area.

In this paper we contribute to the state of the art with compressed and fully-compressed conversions between various formats, all of which then use space *linear in the input plus the output*, and work correctly with high probability:

1. A compressed conversion from any SLP to the LZ parse in $O(n \log^2 n)$ time.
2. A fully-compressed conversion from any SLP to the LZ parse in $O(gz \log^2(n/z))$ time.
3. A compressed conversion from any SLP to a certain (particularly small) LCG [10] in $O(n \log g_{lc})$ time.
4. A fully-compressed conversion from LCGs of some particular kind [10, 27] to the LZ parse in $O(z \log^2(n/z) \log^2 n)$ time.

The third conversion builds a particular LCG whose size is the optimal $O(\delta \log \frac{n}{\delta})$ [27], other similar LCGs [10] can be produced analogously; note that there is a fully-compressed conversion from SLP to LCG [18]; it is for a different LCG, though, and it is not clear, whether it generalizes to other LCGs within given bounds. Also, while the running time of our fourth conversion is larger than Nishimoto et al. [36], we work with a particular LCG, which can be up to $\log n$

Fully compressed conversions			
From \ To	LZ	SLP	LCG
LZ		$O(z\log(n/z))$ [38, 15]	$O(z\log^2 n)$ expected [30] $O(z\log n\log^* n)$ w.h.p. [1] $O(z\log^2(n/z))$ [18]
SLP	$O(g\log^4 n)$ [23] $\boldsymbol{O(gz\log^2(n/z))}$ **w.h.p.**		$O(g\log(n/g))$ [18]
LCG	$O(g_{lc}\log^4 n)$ [23] $O(z\log g_{lc}\log^3 n(\log^* n)^2)$ [36] $\boldsymbol{O(z\log^2(n/z)\log^2 n)}$ **w.h.p.**		

Compressed conversions		
From \ To	LZ	LCG
SLP	$O(n(\log\log n)^2 + z\log^4 n(\log^* n)^2)$ [36] $O(n\sqrt{\log z + \log\log n} + z\log^4 n(\log^* n)^2)$ [36] $\boldsymbol{O(n\log^2 n)}$ **w.h.p.**	$\boldsymbol{O(n\log g_{lc})}$ **w.h.p.**
LCG	$\boldsymbol{O(n\log^2 n)}$ **w.h.p.**	

Table 1. The running times of compressed and fully compressed conversions between LZ, SLP, and LCG, with our contributions in bold. The bounds of Nishimoto et al. [36] are slightly simplified.

times smaller than the LCG of Mehlhorn et al. [30] (we use LCG with a bound of $O(\delta\log\frac{n}{\delta})$, while the latter is only known to be $O(z\log n\log^* n)$ and the only bound on z in terms of δ is $O(\delta\log\frac{n}{\delta})$ [28]). Our contributions together with previously known conversions are depicted in Table 1.

2 Preliminaries

A *string* $T[1..n]$ is a sequence of symbols $T[1]\,T[2]\ldots T[n]$ over an ordered alphabet Σ. For every $1 \le i,j \le n$, $T[1..i] = T[..i]$ is a *prefix* of T, $T[j..n] = T[j..]$ is a *suffix* of T, and $T[i..j]$ is a *substring* of T, which is the empty string ε if $i > j$. The *length* of $T[1..n]$ is $|T| = n$; the length of ε is $|\varepsilon| = 0$. The *concatenation* of two strings $S\cdot S'$ is defined as $S[1]\,S[2]\ldots S[|S|]\,S'[1]\,S'[2]\ldots S'[|S'|]$. The *lexicographic* order between strings $S \ne S'$ is defined as that between $S[1]$ and $S'[1]$ if these are different, or as the lexicographic order between $S[2..]$ and $S'[2..]$ otherwise; the empty string ε is smaller than every other string. The *co-lexicographic* order is defined as the lexicographic order between the reversed strings.

The *Karp-Rabin fingerprint* or the *Karp-Rabin hash* of a string $S[1..n]$ is a value $\phi(S) = \sum_{i=1}^{n}\left(S[i]x^i\right) \bmod p$, for a prime p and $x < p$ [21]. The crucial property of this hash is that if $X \ne Y$, then $\phi(X) \ne \phi(Y)$ with high probability. Another well-known and useful property is that for strings S, S', S' for which $S = S'\cdot S''$ holds, we can compute the hash of any of the strings knowing the hashes of the other two, in $O(1)$ time (see, e.g., [35]).

A *straight-line program (SLP)* of a text T is a context-free grammar in Chomsky normal form (so, in particular, each rule is at most binary) generating only

T, which contains exactly one rule for each nonterminal and the rules can be linearly ordered, such that for any rule $X \rightarrow YZ$ it holds that the rules for both Y and Z precede the rule for X in the ordering.

The height of the SLP is the height of the derivation tree, i.e. the height of a letter is 0 and the height of a nonterminal X with a (unique) rule $X \rightarrow YZ$ is 1 plus maximum of height of Y and Z. The size of the SLP is the number of its rules. We define the *expansion* of a nonterminal X as the string it produces: $\exp(a) = a$ if a is a terminal symbol, and $\exp(X) = \exp(Y) \cdot \exp(Z)$ if $X \rightarrow YZ$.

We say that a grammar is a *locally consistent grammar (LCG)* if it is constructed by iteratively applying rounds of a particular locally consistent parsing, which guarantees that matching fragments $S[i..j] = S[i'..j']$ are parsed the same way, apart from the $O(1)$ blocks from either end. This key property is lifted to such grammars, for which matching fragments are spanned by almost identical subtrees of the parse tree, differing in at most $O(1)$ flanking nonterminals at each level [16, 10]. Such a parsing is defined in Section 5.

The Lempel-Ziv (LZ) parse of a string T [29] is a sequence $F_1, F_2, \ldots, F_z$ of phrases, such that $F_1 \cdot F_2 \cdots F_z = T[1..n]$ and F_i is either a single letter, when this letter is not present in $F_1 \cdot F_2 \cdots F_{i-1}$, or else F_i is the maximal string that occurs twice in $F_1 \cdot F_2 \cdots F_i$, that is, it has an occurrence starting within $F_1 \cdot F_2 \cdots F_{i-1}$; in non-overlapping LZ we additionally require that F_i occurs within $F_1 \cdot F_2 \cdots F_{i-1}$. It is known that $z \leq g = O(z \log(n/z))$, where g is the size of the smallest grammar generating T [38, 8, 15].

We assume the standard word-RAM model of computation with word length $\Theta(\log n)$, in which basic operations over a single word take constant time. Some of our results hold with high probability (w.h.p.), meaning with probability over $1 - n^{-c}$ for any desired constant c. We can make the constant arbitrarily large at the cost of increasing the constant multiplying the running time.

3 Building the LZ parse from an SLP in $\tilde{O}(n)$ time

Our first result computes the LZ parse of a text $T[1..n]$ given an arbitrary SLP of size g that represents T, in time $O(n \log^2 n)$ and space $O(g)$; note that the classic LZ constructions use suffix trees or arrays and use $\Omega(n)$ space. We first describe a couple of tools we need to build on the SLP before doing the conversion.

Lemma 1. *Given an SLP of size g for $T[1..n]$ we can construct in $O(g)$ time and space a new SLP G, and augment it with a data structure such that:*

- *G has height $O(\log n)$.*
- *Any $T[i]$ can be accessed in $O(\log n)$ time.*
- *The Karp-Rabin fingerprint of any $T[i..j]$ can be computed in $O(\log n)$ time.*
- *The longest common prefix of any $T[i..j]$ and $T[i'..j']$ can be computed (w.h.p.) in $O(\log^2 n)$ time.*
- *Any $T[i..j]$ and $T[i'..j']$ can be compared lexicographically and co-lexicographically (w.h.p.) in $O(\log^2 n)$ time.*

Proof. Assume that we are given an SLP with g rules for a text $T[1..n]$. Ganardi et al. [14] showed that in $O(g)$ time and space we can turn it into an SLP G of size $O(g)$ and height $O(\log n)$ and augment G with $O(g)$-space structures that, in $O(\log n)$ time, finds any character $T[i]$ and returns the Karp-Rabin hash of any substring $T[i..j]$ (see Ganardi et al. [14], which refers to a simple data structure from Bille et al. [6]). We work with such augmented G from now on. Given two substrings of T, we can then compute their longest common prefix in $O(\log^2 n)$ time—w.h.p. of obtaining the correct answer—by exponentially searching for its length ℓ [6, Thm. 3]; by checking their characters at offset $\ell + 1$ we can also compare the substrings of T lexicographically within the same time complexity. We can similarly compute the longest common suffix of two substrings and thus compare them co-lexicographically (by comparing the preceding characters). ☐

We will also use a variant of a z-fast trie.

Lemma 2. *Let $\mathcal{S}$ be a lexicographically sorted multiset of m strings of total length n. Then one can build, in $O(n)$ time w.h.p., a data structure of size $O(m)$ that, given a string P, finds in $O(f_h \log |P|)$ time the lexicographic range of the strings in $\mathcal{S}$ prefixed by P, where f_h is the time to compute a Karp-Rabin fingerprint of a substring of P. If this range is nonempty, the answer is correct w.h.p.; if this range is empty, there are no guarantees on correctness of the answer, i.e. the answer could be incorrect.*

Proof. The structure is the z-fast trie of Belazzougui et al. [3, Thm. 5], and the query is the *fat binary search*. A simpler construction was given by Kempa and Kosolobov [24], and it was then fixed, and its construction analyzed, by Navarro and Prezza [35, Sec. 4.3]. ☐

We will resort to a classic grammar-based indexing method [11], for which we need a few definitions and properties.

Definition 1. *The* grammar tree *of an SLP G is formed by pruning the parse tree, converting to leaves, for every nonterminal X, all the nodes labeled X but the leftmost one. An occurrence of a string P in T is* primary *if it spans more than one leaf in the grammar tree; otherwise it is contained in the expansion of a leaf and is* secondary. *If a primary occurrence of P occurs in $\exp(X)$, with rule $X \to YZ$, starting within $\exp(Y)$ and ending within $\exp(Z)$, then the position $P[j]$ aligning to the last position of $\exp(Y)$ is the* splitting point *of the occurrence.*

A small exception to this definition is that, if $|P| = 1$, we say that its primary occurrences are those where it appears at the end of $\exp(X)$ in any leaf X of the grammar tree. We now give a couple of results on primary occurrences.

Lemma 3 ([11]). *A pattern occurring in T has at least one primary occurrence.*

Observation 4 *If X is the lowest nonterminal containing a primary occurrence of P with splitting point j, then, by the way we form the grammar tree, this is the leftmost occurrence of P under X with splitting position j.*

The index sorts all rules $X \rightarrow YZ$ twice: once by the lexicographical order of $\exp(Z)$, while collecting those expansions in a multiset $\mathcal{Z}$, and once by the co-lexicographical order of $\exp(Y)$, while collecting the reversed expansions in a multiset $\mathcal{Y}$. It builds separate z-fast tries (Lemma 2) on $\mathcal{Y}$ and $\mathcal{Z}$, and creates a discrete $g \times g$ grid $\mathcal{G}$, where the cell (x, y) stores the position p iff the xth rule $X \rightarrow YZ$ in the first order is the yth rule in the second order, and $T[p]$ is aligned to the last symbol of $\exp(Y)$ within the occurrence of X as an internal node in the grammar tree. The grid supports orthogonal range queries. The key idea of the index is that, given a search pattern P, for every $1 \leq j \leq |P|$, the lexicographic range $[y_1, y_2]$ of $P[j+1..]$ in $\mathcal{Z}$ and the lexicographic range $[x_1, x_2]$ of the reverse of $P[..j]$ in $\mathcal{Y}$, satisfy that there is a point in the range $[x_1, x_2] \times [y_1, y_2]$ of $\mathcal{G}$ per primary occurrence of P in T with splitting point $P[j]$. The structure $\mathcal{G}$ can determine if the area is empty, or else return a point in it, in time $O(\log g)$. We now build our first tool towards our goal.

Lemma 5. *Given an SLP of size g generating string $T[1..n]$ we can, in space $O(g)$ and time $O(n + g \log g)$ construct w.h.p. a data structure that, given $1 \leq i \leq j \leq k \leq n$, in $O(\log n \log(k-i) + \log^{1+\varepsilon} g)$ time finds w.h.p. the leftmost occurrence of $T[i..k]$ in T that is a primary occurrence with splitting point $T[j]$.*

Proof. We build the components $\mathcal{Y}$, $\mathcal{Z}$, and $\mathcal{G}$ of the described index, following the approach in (see [35, Sec. 4.3–4.4]), all time and space complexities are given there. The data structure is correct w.h.p.. We sort w.h.p. the sets $\mathcal{Y}$ and $\mathcal{Z}$ in $O(g)$ space and $O(n)$ time ([17]), we build the z-fast tries in $O(g)$ space and time $O(n)$ (Lemma 2), and we build the grid data structure in $O(g)$ space and $O(g\sqrt{\log g})$ time ([35, Sec. 4.4], [4]). We note that, by using Lemma 1, we can also do the sorting correctly w.h.p. in $O(g)$ space and $O(g \log g \cdot \log^2 n)$ time.

We use those structures to search for $P = T[i..k]$ with splitting point $T[j]$, that is, we search the z-fast trie of $\mathcal{Z}$ for $T[j+1..k]$ and the z-fast trie of $\mathcal{Y}$ for $T[i..j]$ reversed, in time $O(f_h \log |P|)$; recall Lemma 2. Since the substrings of $T[i..j]$ are also substrings of T, we can compute the Karp-Rabin hash of any substring of $T[i..j]$ in time $f_h = O(\log n)$ by Lemma 1, so this first part of the search takes time $O(\log n \log(k-i))$. Recall from Lemma 2 that this search yields correct results w.h.p., unless the ranges sought are empty, in which case there are no guarantees on correctness.

We now use $\mathcal{G}$ to determine if there are points in the corresponding area. If there are none, then w.h.p. $T[i..k]$ does not occur in T with splitting point $T[j]$. If there are some, then we obtain the value p associated with any point in the range, and compare the Karp-Rabin hash of $T[p-(j-i)..p+(k-j)]$ with that of $T[i..k]$. If they differ, then $T[i..k]$ has no occurrences with splitting point $T[j]$; otherwise w.h.p. the z-fast tries gave the correct range and there are occurrences. This check takes $O(\log n)$ time.

Once we know that (w.h.p.) there are occurrences with splitting point $T[j]$, we want the leftmost one. Each point within the grid range may correspond to a different rule $X \rightarrow YZ$ that splits $T[i..k]$ at $T[j]$; therefore, by Observation 4, we want the minimum of the p values stored for the points within the range. This kind of two-dimensional range minimum query can be solved in time $O(\log^{1+\varepsilon} g)$ and $O(g)$ space, for any constant $\varepsilon > 0$, with an enhancement of $\mathcal{G}$ that uses $O(g)$ space and can be built in time $O(g \log g)$ [31, 9]. This completes the query. $\square$

Finally, we will need the following observation on the monotonicity of occurrences in T, even when we stick to some splitting point.

Observation 6 *If P has a primary occurrence in T with splitting point $P[j]$, then any prefix $P' = P[..k]$, for any $j < k < |P|$, also has a primary occurrence with splitting point $P'[j]$.*

Proof. Let the primary occurrence of P appear in $\exp(X)$ and the occurrence start in $\exp(Y)$ and end in $\exp(Z)$, with $P[j]$ aligned to the last position of $\exp(Y)$. Then $P' = P[..k]$ satisfies the same conditions: a primary occurrence of P' with splitting point $P'[j]$ starts at the same text position. $\square$

We are now ready to give the final result.

Theorem 1. *Given an SLP with g rules for a text $T[1..n]$, w.h.p. we can build the LZ parse of T in $O(n \log^2 n)$ time and within $O(g)$ space.*

Proof. We first build the data structures of Lemma 5 in $O(n + g \log g)$ time and $O(g)$ space, correctly w.h.p.. We then carry out the LZ parse by sliding three pointers left-to-right across T, $i \le j \le k$, as follows: suppose that the parse for $T[1..i-1]$ is already constructed, so a new phrase must start at i. We first check whether $T[i]$ appeared already in $T[1..i-1]$,[8] if not then we create a one-letter phrase and proceed to $i+1$.

If $T[i]$ has occurred earlier, we start the main process of building the next phrase. The invariant is that we have found $T[i..k]$ starting before i in T with splitting point $T[j]$, and there is no primary occurrence of $T[i..k]$ (nor of $T[i..k']$ for any $k' > k$, by Observation 6) with a splitting point in $T[i..j-1]$. To establish the invariant, we initialize j to i and try k from i onwards, using Lemma 5 and advancing k as long as the leftmost occurrence of $T[i..k]$ with splitting point $T[i]$ starts to the left of i.

Note that we will succeed the first time, for $k = i$. We continue until we reach $k = n$ (and output $T[i..n]$ as the last phrase of the LZ parse) or we cannot find $T[i..k+1]$ starting before i with splitting point $T[i]$. We then try successive values of j, from $i+1$ onwards, using Lemma 5 to find $T[i..k+1]$ starting before i with splitting point $T[j]$. If we finally succeed for some $j \le k$, we reestablish the invariant by increasing k and return to the first loop, which again increases k with fixed j, and so on.

When j reaches $k+1$, it follows that $T[i..k]$ occurs before i and $T[i..k+1]$ does not, with any possible splitting point. The next phrase is then $T[i..k]$, which we output, reset $i = k+1$, and resume the parsing.

[8] This is easily done in $O(1)$ time and $|\Sigma| \in O(g)$ space by just storing an array with the leftmost occurrence of every distinct symbol in T. This array is built in $O(g)$ time from the leaves of the grammar tree.

Since j and k never decrease in the process, we use queries from Lemma 5 $O(n)$ times for a total time of $O(n + g \log g + n(\log^2 n + \log^{1+\varepsilon} g)) = O(n \log^2 n)$ to build the LZ parse. $\square$

4 Building the LZ parse from an SLP in $\tilde{O}(gz)$ time

If T is highly compressible, the running time $O(n \log^2 n)$ in Theorem 1 could be exponential in the size $O(g)$ of the input. We can build the parse in $O(gz \log^2 \frac{n}{z}) \subset$ poly(g) time by using, instead of the machinery of the preceding section, Jeż's [19] algorithm for fully-compressed pattern matching. We will only balance the SLP if needed [14] so that its height is $O(\log n)$. We start by reminding some tools.

Lemma 7 ([38]). *Given an SLP of height h for T, we can in $O(h)$ time and space produce an SLP of size $O(h)$ for any desired substring $T[i..j]$ (without modifying the SLP of T).*

Note that the SLP constructed in the Lemma above may use some of the nonterminals of the original SLP for T, i.e. its size is in principal $g + O(h)$.

Lemma 8 ([19]). *If T and P have SLPs of size g and g', then we can find the leftmost occurrence of P in T in time $O((g + g') \log |P|)$, within $O(g + g')$ space.*

Note that [19] does not state the space complexity, however, the analysis [19, Sec. 6] bounds intermediate SLPs to be of size $O(g + g')$ ([19, Lem. 6.5] and the running time of the subprocedures (and so their space usage) to be linear; hence the linear space consumption follows.

Assume again we have already parsed $T[1..i-1]$, and aim to find the next phrase, $T[i..k]$. We will exponentially search for k using $O(\log(k-i))$ steps. Each step implies determining whether some $T[i..j]$ occurs in T starting to the left of i (so that k is the maximum such j). To do this we exploit the fact that our SLP is of height $h = O(\log n)$ and use Lemma 7 to extract an SLP for $T[i..j]$, of size $g' \leq g + O(h) = g + O(\log n)$, in $O(h) = O(\log n)$ time[9]. We then search for the SLP of $T[i..j]$ in the SLP of size g of T using Lemma 8, in time $O((g + g') \log(j - i)) \subseteq O(g \log(k - i))$ (because $g' \subseteq O(g)$, as g is always $\Omega(\log n)$). By comparing the leftmost occurrence position with i we drive the exponential search, finding k in time $O(g \log^2(k - i))$ and space $O(g)$.

Repeating this for each LZ phrase we get $\sum_{i=1}^{z} g \log^2 n_i$, where $n_1, n_2, \ldots, n_z$ denote the consecutive phrase lengths. By Jensen's inequality (since $\log^2(\cdot)$ is concave), the sum is maximized when all $n_i = n/z$.

Theorem 2. *Given an SLP with g rules for a text $T[1..n]$ whose LZ parse has z phrases, we can build that parse in $O(gz \log^2(n/z))$ time and $O(g)$ space.*

[9] Rytter [38] rebalances the grammar he extracts, but we do not need to do this.

5 Building an LCG from an SLP in $\tilde{O}(n)$ time

Locally consistent grammars (LCGs) are actually run-length context-free grammars, that is, they allow rules $X \rightarrow Y_1 \cdots Y_t$ (of size t) and run-length rules of the form $X \rightarrow Y^t$, equivalent to $X \rightarrow Y \cdots Y$ (t copies of Y), of size 2. A particular kind of LCG can be obtained from T with the following procedure [27]. First, define $\ell_k = (4/3)^{\lceil k/2 \rceil - 1}$ and call $S_0 = T$. Then, for increasing levels $k > 0$, create S_k from S_{k-1} as follows:

1. If k is odd, find the maximal runs of (say, $t > 1$ copies of) equal symbols Y in S_{k-1} such that $|\exp(Y)| \leq \ell_k$, create a new grammar rule $X \rightarrow Y^t$, and replace the run by X. The other symbols are copied onto S_k as is.
2. If k is even, generate a function π_k that randomly reorders the symbols of S_{k-1} and define local minima as the positions $1 < i < |S_{k-1}|$ such that $\pi_k(S_{k-1}[i-1]) > \pi_k(S_{k-1}[i]) < \pi_k(S_{k-1}[i+1])$. Place a block boundary after each local minimum, and before and after the symbols Y with $|\exp(Y)| > \ell_k$. Create new rules for the resulting blocks of length more than 1 and replace them in S_k by their corresponding nonterminals. Leave other symbols as is.

Our plan is to extract T left to right from its SLP, in $O(n)$ time, and carry out the described process in streaming form. The only obstacle to perform the process at level k in a single left-to-right pass is the creation of the functions π_k without knowing in advance the alphabet of S_{k-1}. We can handle this by maintaining two balanced trees. The first, T_{id}, is sorted by the actual symbol identifiers, and stores for each symbol a pointer to its node in the second tree, T_{pos}. The tree T_{pos} is sorted by the current π_k values (which evolve as new symbols arise), that is, the π_k value of a symbol is its inorder position in T_{pos}. We can know the current value of a symbol in π_k by going up from its node in T_{pos} to the root, adding up one plus the number of nodes in the left subtree of the nodes we reach from their right child (so T_{pos} stores subtree sizes to enable this computation). Two symbols are then compared in logarithmic time by computing their π_k values using T_{pos}.

When the next symbol is not found in T_{id}, it is inserted in both trees. Its rank r in T_{pos} is chosen at random in $[1, |T_{id}|+1]$. We use the subtree sizes to find the insertion point in T_{pos}, starting from the root: let t_l be the size of the left child of a node. If $r \leq t_l + 1$ we continue by the left child, otherwise we subtract $t_l + 1$ from r and continue by the right child. The balanced tree rotations maintain the ranks of the nodes, so the tree can be rebalanced after the insertion adds a leaf.

Our space budget does not allow us maintaining the successive strings S_k. Rather, we generate $S_0 = T$ left to right in linear time using the given SLP and have one iterator per level k (the number of levels until having a single nonterminal is logarithmic [27, Remark 3.16]). Each time the process at some level $k-1$ produces a new symbol, it passes that new symbol on to the next level, k. When the last symbol of T is consumed, all the levels in turn close their processes, bottom-up; the LCG comprises the rules produced along all levels.

The total space used is proportional to the number of distinct symbols across all the levels of the grammar. This can be larger than the grammar size because symbols X with $|\exp(X)| > \ell_k$ are not replaced in level k, so they exist in the next levels as well. To avoid this, we perform a twist that ensures that every distinct grammar symbol is stored only in $O(1)$ levels. The twist is not to store in the trees the symbols that cannot form groups in this level, that is, those X for which $|\exp(X)| > \ell_k$. Since then the symbols stored in the tree for even levels k are forced to form blocks (no two consecutive minima can exist), they will no longer exist in level $k+1$. Note that the sizes of the trees used for the symbols at level k are then proportional to the number of nonterminals of that level in the produced grammar.

There is a deterministic bound $O(\delta \log \frac{n}{\delta})$ on the total number of nonterminals in the generated grammar [27, Corollary 3.12], and thus on the total sizes of the balanced trees. Here, δ is the compressibility measure based on substring complexity, and size $O(\delta \log \frac{n}{\delta})$ is optimal for every n and δ [27, 28]. The size g_{lc} of the produced LCG could be higher, as for some choices of letter permutations on various levels some right-hand of the productions can be of not-constant length, however, but it is still $O(\delta \log \frac{n}{\delta})$ in expectation and with high probability [27, Theorem 3.13]. Because the sum of the lengths of the strings S_k is $O(n)$ [27, Corollary 3.15], we produce the LCG in time $O(n \log g_{lc})$; the $\log g_{lc}$ comes from the cost of balanced tree operations.

Theorem 3. *Given an SLP with g rules for a text $T[1..n]$, we can build w.h.p. an LCG of size $g_{lc} = O(\delta \log \frac{n}{\delta})$ for T in $O(n \log g_{lc})$ time and $O(g + g_{lc})$ space.*

If we know δ, we can abort the construction as soon as its total size exceeds $c \cdot \delta \log \frac{n}{\delta}$ for some suitable constant c, and restart the process afresh. After $O(1)$ attempts in expectation, we will obtain a locally consistent grammar of size $O(\delta \log \frac{n}{\delta})$ [27, Corollary 3.15]. The grammar we produce, in $O(n \log g_{lc})$ expected time, is then of guaranteed size $g_{lc} = O(\delta \log \frac{n}{\delta})$. Note that we need a structure mapping blocks and runs to new symbols: using a simple trie for the rules uses $O(g_{lc})$ space and can be constructed in $O(g_{lc} \log g_{lc})$ time.

6 Building the LZ parse from an LCG in $\tilde{O}(z)$ time

One of the many advantages of LCGs compared to general SLPs is that, related to Definition 1, they may allow trying out only $O(\log |P|)$ splitting positions of P in order to discover all their primary occurrences, as opposed to $m-1$ if using a generic SLP. This is the case of the LCG of size $O(\delta \log \frac{n}{\delta})$ of the previous section [27], which specializes [10], in the sense that any grammar produced with the first method [27] can be produced by the second [10], and therefore every property we prove for the second method holds for the first as well. The first method introduces a restriction to produce grammars of size $O(\delta \log \frac{n}{\delta})$, whereas the second kind has a weaker space bound of $O(\gamma \log \frac{n}{\gamma})$, where $\gamma \geq \delta$ is the size of the smallest string attractor of T [25] (concretely, the parsing is as in Section 5 but does not enforce the condition $\exp(X) \leq \ell_k$). We now show how the bound on the splitting positions number enables us to find the LZ parse of those LCGs in time $O(z \log^4 n)$. We will then stick to the more general LCG [10]; the results hold for the other too [27], as explained.

Our technique combines results used for Theorems 1 and 2: we will use exponential search, as in Section 4, to find the next phrase $T[i..k]$, and will use the data structures of Section 3 to search for its leftmost occurrence in T; the fact that we will need to check just a logarithmic number of splitting points will yield the bound. We start with an analogue of Lemma 1 for our LCG; we get better bounds in this case.

Lemma 9. *Given the LCG [10] of size g_{lc} of $T[1..n]$, we can build in $O(g_{lc} \log g_{lc})$ time and $O(g_{lc})$ space a data structure supporting the same operations listed in Lemma 1, all in $O(\log n)$ time.*

Proof. Since the LCG is already balanced, accessing $T[i]$ in $O(\log n)$ time is immediate. The Karp-Rabin fingerprints can be computed with the structure of Christiansen et al. [10, Thm. A.3], which can be built in $O(g_{lc})$ space and time.

To compute longest common prefixes (LCPs) we use a similar approach as Kempa and Kociumaka [23, Thm. III.3] or earlier Alstrup, Brodal and Rauhe [1]. To deal with rules of non-constant size, we build a data structure for answering the LCE queries on the (right-hand sides of) non-runs *rules* of the LCG. This is a standard construction (using suffix arrays and LCA queries [5]) and can be done in $O(g_{lc} \log g_{lc})$ time and $O(g_{lc})$ space, or even in $O(g_{lc})$ time, when the letters can be identified with numbers that are polynomial in g_{lc} [20]. $\square$

Consider the cost to build the data structures of Section 3. Using Lemma 9, we sort the multisets $\mathcal{Y}$ and $\mathcal{Z}$ in time $O(g_{lc} \log g_{lc} \cdot \log n)$. This time dominates the construction time of the z-fast tries for $\mathcal{Y}$ and $\mathcal{Z}$, the grid structure $\mathcal{G}$, and the two-dimensional range minimum query mentioned in Lemma 5. Further, because $g_{lc} \le \gamma \log \frac{n}{\gamma}$ [10] and $\gamma \le z$ [25], this time is in $O(z \log^2 n \log(n/z))$.

After building those components, we start parsing the text using the exponential search of Section 4. To test whether the candidate phrase $T[i..j]$ occurs starting to the left of i, we use the LCG search algorithm for $T[i..j]$ provided by the LCG. Christiansen et al. [10] observed that we need to check only $O(\log(j-i))$ splitting points to find every primary occurrence of $T[i..j]$. They find the splitting points through a linear-time parse of $T[i..j]$, but we can do better by reusing the locally consistent parsing used to build the LCG. While we do not store the strings S_k of Section 5, we can recover the pieces that cover $T[i..j]$ by traversing the (virtual) grammar tree from the root towards that substring of T.

Lemma 10 ([10]). *Let $M_0(i,j) = \{i, j-1\}$. For any $k > 0$, let $M_k(i,j)$ contain the first and last positions ending a block of S_k that are within $T[i..j-1]$ but do not belong to $M_{k'}(i,j)$ for any $k' < k$. Then, $M(i,j) = \cup_k M_k(i,j)$ is of size $O(\log(j-i))$ and the splitting point of every primary occurrence of $T[i..j]$ in T belongs to $M(i,j)$.*

Proof. Our definition of $M(i,j)$ includes the positions in Definitions 4.7 and 4.8 of Christiansen et al. [10] (they use B_r and $\hat{B}_r$ instead of our even and odd levels S_k). The property we state corresponds to their Lemma 6.4 [10]. $\square$

To compute $M(i,j)$, then, we descend from the root of the (virtual) parse tree of the LCG towards the lowest nonterminal X that fully contains $T[i..j]$, and continue from X towards the leaf L that contains $T[i]$. We then start adding to $M(i,j)$ the endpoint of L (which is i), and climb up to its parent P. If P ends in the same position of L, we shift P to its next sibling. We now set $L = P$, add the last position of L to $M(i,j)$, climb up to its parent P, and so on until the last position of L exceeds $j-1$ (which may occur when reaching X or earlier). We proceed analogously with the path from X to the leaf that contains $T[j-1]$.

We visit $O(\log n)$ nodes in this process, but since the LCG may not be binary, we may need $O(\log n)$ time to find the proper children of a node. The total time is then $O(\log^2 n)$.

Once the set $M(i,j)$ of splitting points is found, we search for each of them as in Lemma 5, each in time $O(\log(j-i)\log n + \log^{1+\varepsilon} g_{lc})$. Therefore, the total time to check a candidate $T[i..j]$ is $O\left(\log^2 n + \log(j-i)\left(\log(j-i)\log n + \log^{1+\varepsilon} g_{lc}\right)\right)$. In turn, the exponential search that finds the next phrase $T[i..k]$ carries out $O(\log(k-i))$ such checks, with $j - i \le 2(k-i)$, thus the total time to find the next phrase is $O(\log(k-i)\log^2 n + \log^3(k-i)\log n + \log^2(k-i)\log^{1+\varepsilon} g_{lc})$. Using Jensen's inequality again and simplifying, this yields the running time of $O(z\log^2(n/z)\log^2 n)$.

Theorem 4. *Given the LCG of Christiansen et al. [10] of size g_{lc} of $T[1..n]$, we can build w.h.p. the LZ parse of T in $O(z\log^2(n/z)\log^2 n)$ time and $O(g_{lc})$ extra space. The result also holds verbatim for the LCG of Kociumaka et al. [27].*

7 Conclusions

We have contributed to the problem of *compressed conversions*, that is, using asymptotically optimal space, between various compression formats for repetitive data. Such a space means linear in the input plus output size, which outrules the possibility of decompressing the data. This is crucial to face the sharp rise the size of data in sequence form has experienced in the last decades, which requires manipulating the data always in compressed form. To the best of our knowledge, we are the first to propose methods to build the Lempel-Ziv parse of a text directly from its straight-line program representation. Our methods work in time $O(n\log^2 n)$ and $O(gz\log^2 n)$. The second is polynomial on the size of the compressed data and we thus call it a *fully-compressed conversion*; such methods can be considerably faster when the data is highly compressible. We also gave methods to convert from straight-line programs to locally consistent grammars, which enable faster and more complex queries, in $O(n\log n)$ time. As a showcase for their improved search capabilities, we show how to produce the Lempel-Ziv parse from those grammars in time $O(z\log^4 n)$, another fully-compressed conversion. All of our conversions work with high probability.

Obvious open problems are obtaining better running times without using more space. Furthermore, we think that approaches similar to those described in this article can be applied to effectively compute other parses, such as the lexicographic parse [34]. We plan to address these in the extended version.

Acknowledgements. We thank the anonymous reviewers, whose comments helped to improve the presentation, correct several errors and give a much better and more detailed exhibition of the state of the art in the introduction.

References

1. S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828, 2000.
2. H. Arimura, S. Inenaga, Y. Kobayashi, Y. Nakashima, and M. Sue. Optimally computing compressed indexing arrays based on the compact directed acyclic word graph. In *International Symposium on String Processing and Information Retrieval*, pages 28–34, 2023.
3. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *18th European Symposium on Algorithms (ESA), Part I*, pages 427–438, 2010.
4. D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *27th Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.
5. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.
6. P. Bille, I. L. Gørtz, P. H. Cording, B. Sach, H. W. Vildhøj, and S. Vind. Fingerprints in compressed strings. *J. Comput. Syst. Sci.*, 86:171–180, 2017.
7. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.
8. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
9. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
10. A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):article 8, 2020.
11. F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.
12. R. M. Durbin, A. Auton, and L. D. Brooks. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.
13. T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.
14. M. Ganardi, A. Jeż, and M. Lohrey. Balancing straight-line programs. *J. ACM*, 68(4):27:1–27:40, 2021.
15. P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *European Symposium on Algorithms*, pages 421–432, 2011.
16. P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. In *29th ACM-SIAM Symposium on Discrete Algorithms*, pages 1509–1528, 2018.
17. P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *28th ACM-SIAM Symposium on Discrete Algorithms*, pages 425–439. SIAM, 2017.

18. T. I. Longest common extensions with recompression. In *28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 18:1–18:15, 2017.
19. A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms (TALG)*, 11(3):1–43, 2015.
20. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
21. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.
22. D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *61st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020.
23. D. Kempa and T. Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *63rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1877–1886, 2023.
24. D. Kempa and D. Kosolobov. LZ-End parsing in compressed space. In *27th Data Compression Conference (DCC)*, pages 350–359, 2017.
25. D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *50th ACM SIGACT Symposium on Theory of Computing*, pages 827–840, 2018.
26. J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.
27. T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ-optimal space. *Algorithmica*, 2023. accepted, available online.
28. T. Kociumaka, G. Navarro, and N. Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023.
29. A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
30. K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
31. G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
32. G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.
33. G. Navarro. Computing MEMs on repetitive text collections. In *34th Symposium on Combinatorial Pattern Matching (CPM)*, page article 22, 2023.
34. G. Navarro, C. Ochoa, and N. Prezza. On the approximation ratio of ordered parsings. *IEEE Transactions on Information Theory*, 67(2):1008–1026, 2020.
35. G. Navarro and N. Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.
36. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. *Discret. Appl. Math.*, 274:116–129, 2020.
37. A. Policriti and N. Prezza. From LZ77 to the run-length encoded Burrows-Wheeler Transform, and back. In *28th Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 17:1–17:10, 2017.
38. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.