Optimal-Time Dictionary-Compressed Indexes

Anders Roy Christiansen, The Technical University of Denmark, Denmark
Mikko Berggren Ettienne, The Technical University of Denmark, Denmark
Tomasz Kociumaka, Bar-Ilan University, Israel, and University of California, Berkeley, US
Gonzalo Navarro, CeBiB and University of Chile, Chile
Nicola Prezza, Ca' Foscari University of Venice, Italy

We describe the first self-indexes able to count and locate pattern occurrences in optimal time within a space bounded by the size of the most popular dictionary compressors. To achieve this result we combine several recent findings, including string attractors — new combinatorial objects encompassing most known compressibility measures for highly repetitive texts —, and grammars based on locally-consistent parsing.

More in detail, let γ be the size of the smallest attractor for a text T of length n. The measure γ is an (asymptotic) lower bound to the size of dictionary compressors based on Lempel–Ziv, context-free grammars, and many others. The smallest known text representations in terms of attractors use space $O(\gamma \log(n/\gamma))$, and our lightest indexes work within the same asymptotic space. Let $\epsilon > 0$ be a suitably small constant fixed at construction time, m be the pattern length, and occ be the number of its text occurrences. Our index counts pattern occurrences in $O(m + \log^{2+\epsilon} n)$ time, and locates them in $O(m + (occ + 1)\log^{\epsilon} n)$ time. These times already outperform those of most dictionary-compressed indexes, while obtaining the least asymptotic space for any index searching within $O((m + occ)\operatorname{polylog} n)$ time. Further, by increasing the space to $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$, we reduce the locating time to the optimal $O(m + occ)$, and within $O(\gamma \log(n/\gamma) \log n)$ space we can also count in optimal $O(m)$ time. No dictionary-compressed index had obtained this time before. All our indexes can be constructed in $O(n)$ space and $O(n \log n)$ expected time.

As a byproduct of independent interest, we show how to build, in $O(n)$ expected time and without knowing the size γ of the smallest attractor (which is NP-hard to find), a run-length context-free grammar of size $O(\gamma \log(n/\gamma))$ generating (only) T. As a result, our indexes can be built without knowing γ.

Additional Key Words and Phrases: Repetitive string collections; Compressed text indexes; Attractors; Grammar compression; Locally-consistent parsing

1. INTRODUCTION

The need to search for patterns in large string collections lies at the heart of many text retrieval, analysis, and mining tasks, and techniques to support it efficiently have been studied

T. Kociumaka was supported by ISF grants no. 1278/16 and 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 683064). G. Navarro was supported by Fondecyt grant 1-200038, Chile, and Basal Funds FB0001, ANID, Chile. N. Prezza was supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant no. RBSI146R5L.
A preliminary version of this article appeared in Proc. LATIN'18 [Christiansen and Ettienne 2018].
Author's addresses: Anders Roy Christiansen, The Technical University of Denmark, Denmark, aroy@dtu.dk. Mikko Berggren Ettienne, The Technical University of Denmark, Denmark, miet@dtu.dk. Tomasz Kociumaka, Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel and IEOR Department, University of California, Berkeley, US, kociumaka@mimuw.edu.pl. Gonzalo Navarro, CeBiB – Center for Biotechnology and Bioengineering, Chile and Department of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl. Nicola Prezza, Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Italy, nicola.prezza@unive.it.

DOI:http://dx.doi.org/10.1145/0000000.0000000

for decades: the suffix tree, which is the landmark solution, is over 40 years old [Weiner 1973; McCreight 1976]. The recent explosion of data in digital form led the research since 2000 towards compressed self-indexes, which support text access and searches within compressed space [Navarro and Mäkinen 2007]. This research, though very successful, is falling short to cope to a new wave of data that is flooding our storage and processing capacity with volumes of higher orders of magnitude that outpace Moore's Law [Stephens et al. 2015]. Interestingly enough, this massive increase in data size is often not accompanied with a proportional increase in the amount of information that data carries: much of the fastest-growing data is highly repetitive, for example thousands of genomes of the same species, versioned document and software repositories, periodic sky surveys, and so on. Dictionary compression of those datasets typically reduces their size by two orders of magnitude [Gagie et al. 2018]. Unfortunately, previous self-indexes build on statistical compression, which is unable to capture repetitiveness [Kreft and Navarro 2013]; therefore, a new generation of compressed self-indexes based on dictionary compression is emerging.

Examples of successful compressors from this family include (but are not limited to) the Lempel–Ziv factorization [Lempel and Ziv 1976], of size z; context-free grammars [Kieffer and Yang 2000] and run-length context-free grammars [Nishimoto et al. 2016], of size g; bidirectional macro schemes [Storer and Szymanski 1982], of size b; and collage systems [Kida et al. 2003], of size c. Other compressors that are not dictionary-based but also perform well on repetitive text collections are the run-length Burrows–Wheeler transform [Burrows and Wheeler 1994], of size ρ, and the CDAWG [Blumer et al. 1987], of size e. A number of compressed self-indexes have been built on top of those compressors; Gagie et al. [2018] give a thorough review.

Recently, Kempa and Prezza [2018] showed that all the above-mentioned repetitiveness measures (i.e., z, g, b, c, ρ, e) are never asymptotically smaller than the size γ of a new combinatorial object called string attractor. This and subsequent works [Kempa and Prezza 2018; Navarro and Prezza 2019; Prezza 2019] showed that efficient access and searches can be supported within $O(\gamma \log(n/\gamma))$ space. By the nature of this new repetitiveness measure, such data structures are universal, in the sense that they can be used on top of a wide set of dictionary-compressed representations of T.

Our results. In this article we obtain the best results on attractor-based indexes, including the first optimal-time search complexities within space bounded in terms of γ, z, g, b, or c. We combine and improve upon three recent results:

(1) Navarro and Prezza [2019, Thm. 2] presented the first index that builds on an attractor of size γ of a text $T[1..n]$. It uses $O(\gamma \log(n/\gamma))$ space and finds the *occ* occurrences of a pattern $P[1..m]$ in time $O(m \log n + occ(\log\log(n/\gamma) + \log^{\epsilon} \gamma))$ for any constant $\epsilon > 0$.
(2) Christiansen and Ettienne [2018, Thm. 2(3)] presented an index that builds on the Lempel–Ziv parse of T, of $z \geq \gamma$ phrases, which uses $O(z \log(n/z))$ space and searches in time[1] $O(m + \log^{\epsilon}(z \log(n/z)) + occ(\log\log n + \log^{\epsilon} z))$.
(3) Navarro [2019, Thm. 5] presented the first index that builds on the Lempel–Ziv parse of T and counts the number of occurrences of P in T (i.e., computes *occ*) in time $O(m \log n + m \log^{2+\epsilon} z)$, using $O(z \log(n/z))$ space.

Our contributions are as follows:

[1]This is the conference version of the present article, where we mistakenly claim a slightly better time of $O(m + \log^{\epsilon} z + occ(\log\log n + \log^{\epsilon} z))$. The error can be traced back to the wrong claim that our two-sided range structure, built on $O(z \log(n/z))$ points, answers queries in $O(\log^{\epsilon} z)$ time (the correct time is, instead, $O(\log^{\epsilon}(z \log(n/z)))$). The second occurrence of $\log^{\epsilon} z$, however, is correct, because the missing term is absorbed by $O(\log\log n)$.

(1) We obtain, in space $O(\gamma \log(n/\gamma))$, an index that lists all the occurrences of P in T in time $O(m + \log^{\epsilon} \gamma + occ \log^{\epsilon}(\gamma \log(n/\gamma)))$, thereby obtaining the best space and improving the time from previous works [Christiansen and Ettienne 2018; Navarro and Prezza 2019].
(2) We obtain, in space $O(\gamma \log(n/\gamma))$, an index that counts the occurrences of P in T in time $O(m + \log^{2+\epsilon}(\gamma \log(n/\gamma)))$, which outperforms the previous result [Navarro 2019] both in time and space.
(3) Using more space, $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$, we list the occurrences in optimal $O(m + occ)$ time, and within space $O(\gamma \log(n/\gamma) \log n)$, we count them in optimal $O(m)$ time.

We can build all our structures in $O(n \log n)$ expected time and $O(n)$ working space, without the need to know the size γ of the smallest attractor.

Our first contribution uses the minimum known asymptotic space, $O(\gamma \log(n/\gamma))$, for any dictionary-compressed index searching in time $O((m + occ) \operatorname{polylog} n)$ [Gagie et al. 2018]. Only recently [Navarro and Prezza 2019], it has been shown that it is possible to search within this space. Indeed, our new index outperforms most dictionary-compressed indexes, with a few notable exceptions like Gagie et al. [2014], who use $O(z \log(n/z))$ space and $O(m \log m + occ \log \log n)$ search time (but, unlike us, assume a constant alphabet), and Bille et al. [2018], who use $O(z \log(n/z) \log \log z)$ space and $O(m + occ \log \log n)$ search time without making any assumption on the alphabet size. Our second contribution lies on a less explored area, since the first index able to count efficiently within dictionary-bounded space is very recent [Navarro 2019].

Our third contribution yields the first indexes with space bounded in terms of γ, z, g, b, or c, multiplied by any $O(\operatorname{polylog} n)$, that searches in optimal time. Such optimal times have been obtained, instead, by using $O(\rho \log(n/\rho))$ space [Gagie et al. 2018], or using $O(e)$ space [Belazzougui and Cunial 2017]. Various experiments [Belazzougui et al. 2015; Gagie et al. 2018], however, show that measures ρ and e are usually considerably larger than z on repetitive texts.

As a byproduct of independent interest, we show how to build a run-length context-free grammar (RLCFG) of size $O(\gamma \log(n/\gamma))$ generating (only) T, where γ is the size of the smallest attractor, in $O(n)$ expected time and without the need to know the attractor. We use this result to show that our indexes do not need to know an attractor, nor its minimum possible size γ (which is NP-hard to obtain [Kempa and Prezza 2018]) in order to achieve their attractor-bounded results. This makes our results much more practical. Another byproduct is the generalization of our results to arbitrary CFGs and, especially, RLCFGs, yielding slower times in $O(g)$ space, which can potentially be $o(\gamma \log(n/\gamma))$.

Techniques. A key component of our result is the fact that one can build a locally-consistent and locally-balanced grammar generating (only) T such that only a few splits of a pattern P must be considered in order to capture all of its "primary" occurrences [Kärkkäinen and Ukkonen 1996]. Previous parsings had obtained $O(\log m \log^{*} n)$ [Nishimoto et al. 2019] and $O(\log n)$ [Gawrychowski et al. 2018] splits, but now we build on a parsing by Mehlhorn et al. [1997] to obtain $O(\log m)$ splits with a grammar of size $O(\gamma \log(n/\gamma))$.

Our first step is to define a variant of Mehlhorn et al.'s randomized parsing and prove, in Section 3, that it enjoys several locality properties we require later for indexing. In Section 4, we use the parsing to build a RLCFG with the local balancing and local consistency properties we need. We then show, in Section 5, that the size of this grammar is bounded by $O(\gamma \log(n/\gamma))$, by proving that new nonterminals appear only around attractor positions.

In that section, we also show that the grammar can be built without knowing the minimum size γ of an attractor of T. This is important because, unlike z, which can be computed in $O(n)$ time, finding γ is NP-hard [Kempa and Prezza 2018]. For this sake we define a new

measure of compressibility, $\delta \leq \gamma$, which can be computed in $O(n)$ time and can be used to bound the size of the grammar.

Section 6 describes our index. We show how to parse the pattern in linear time using the same text grammar, and how to do efficient substring extraction and Karp–Rabin fingerprinting from a RLCFG. Importantly, we prove that only $O(\log m)$ split points are necessary in our grammar. All these elements are needed to obtain time linear in m. We also build on existing techniques [Claude and Navarro 2012] to obtain time linear in *occ* for the "secondary" occurrences; the primary ones are found in a two-dimensional data structure and require more time. Finally, by using a larger two-dimensional structure and introducing new techniques to handle short patterns, we raise the space to $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$ but obtain the first dictionary-compressed index using optimal $O(m + occ)$ time.

In Section 7 we use the fact that only $O(\log m)$ splits must be considered to reduce the counting time of Navarro [2019], while making its space attractor-bounded as well. This requires handling the run-length rules of RLCFGs, which turns out to require new ideas exploiting string periodicities. Further, by handling short patterns separately and raising the space to $O(\gamma \log(n/\gamma) \log n)$, we obtain the first dictionary-compressed index that counts in optimal time, $O(m)$.

Along the article we obtain various results on accessing and indexing specific RLCFGs. We generalize them to arbitrary CFGs and RLCFGs in Appendix A.

An earlier version of this article appeared in Proc. LATIN'18 [Christiansen and Ettienne 2018]. This article is an exhaustive rewrite where we significantly extend and improve upon the conference results. We use a slightly different grammar, which requires re-proving all the results, in particular correcting and completing many of the proofs in the conference paper. We have also reduced the space by building on attractors instead of Lempel–Ziv parsing, used better techniques to report secondary occurrences and handle short patterns, and ultimately obtained optimal locating time. All the results on counting are also new.

2. BASIC CONCEPTS

Strings and texts. A string is a sequence $S[1\,..\,\ell] = S[1]S[2]\cdots S[\ell]$ of symbols. The symbols belong to an alphabet Σ, which is a finite subset of the integers. The length of S is written as $|S| = \ell$.

A string Q is a substring of S if Q is empty or $Q = S[i]\cdots S[j]$ for some indices $1 \leq i \leq j \leq \ell$. The occurrence of Q at position i of S is a fragment of S denoted $S[i\,..\,j]$. We then also say that $S[i\,..\,j]$ matches Q. We assume implicit casting of fragments to the underlying substrings so that $S[i\,..\,j]$ may also denote $S[i]\cdots S[j]$ in contexts requiring strings rather than fragments.

A suffix of S is a fragment of the form $S[i\,..\,\ell]$, and a prefix is a fragment of the form $S[1\,..\,i]$. The juxtaposition of strings and/or symbols represents their concatenation, and the exponentiation denotes the iterated concatenation. The reverse of $S[1\,..\,\ell]$ is $S^{rev} = S[\ell]S[\ell-1]\cdots S[1]$.

We will index a string $T[1\,..\,n]$, called the text. We assume our text to be flanked by special symbols $T[1] = \#$ and $T[n] = \$$ that belong to Σ but occur nowhere else in T. This, of course, does not change any of our asymptotic results, but it simplifies matters.

Karp–Rabin signatures. Karp–Rabin fingerprinting [Karp and Rabin 1987] assigns to every string $S[1\,..\,\ell]$ a signature $\kappa(S) = (\sum_{i=1}^{\ell} S[i] \cdot c^{i-1}) \bmod \mu$ for a suitable integer c and a prime number μ. It is possible to build a signature formed by a pair of functions $\langle \kappa_1, \kappa_2 \rangle$ guaranteeing no collisions between substrings of $T[1\,..\,n]$, in $O(n \log n)$ expected time [Bille et al. 2014].

With high probability. The term with high probability (w.h.p.) means with probability at least $1 - n^{-c}$ for an arbitrary constant parameter c, where n is the input size (in our case, the length of the text).

Model of computation. We use the RAM model with word size $w = \Omega(\log n)$, allowing classic arithmetic and bit operations on words in constant time. Our logarithms are to the base 2 by default.

3. LOCALLY-CONSISTENT PARSING

A string $S[1\,..\,n]$ can be parsed in a locally consistent way, meaning that equal substrings are largely parsed in the same form. We use a variant of the parsing of Mehlhorn et al. [1997].

Let us define a run in a string as a maximal substring repeating one symbol. The parsing proceeds in two passes. First, it groups the runs into metasymbols, which are seen as single symbols. The resulting sequence is denoted $\hat{S}[1\,..\,\hat{n}]$. The following definition describes the process precisely and defines mappings between S and $\hat{S}$.

Definition 3.1. The string $\hat{S}[1\,..\,\hat{n}]$ is obtained from a string $S[1\,..\,n]$ by replacing every distinct run a^ℓ in S by a special metasymbol $\boxed{a^\ell}$ so that two occurrences of the same run a^ℓ are replaced by the same metasymbol. The alphabet $\hat{\Sigma}$ of $\hat{S}$ consists of the metasymbols that represent runs in S, that is $\hat{\Sigma} = \{\boxed{a^\ell} : a^\ell \text{ is a run in } S\}$.

A position $S[i]$ that belongs to a run a^ℓ is mapped to the position $\hat{S}[\hat{i}]$ of the corresponding metasymbol $\boxed{a^\ell}$, denoted $\hat{i} = \mathrm{map}(i)$. A position $\hat{S}[\hat{i}]$ is mapped back to the maximal range $\mathrm{map}^{-1}(\hat{i}) = [\mathrm{map}^{-\mathrm{F}}(\hat{i})\,..\,\mathrm{map}^{-\mathrm{L}}(\hat{i})]$ of positions in S that map to $\hat{i}$. That is, if $S[i\,..\,i+\ell-1]$ is a run in S that maps to $\hat{i}$, then $\mathrm{map}^{-\mathrm{F}}(\hat{i}) = i$ and $\mathrm{map}^{-\mathrm{L}}(\hat{i}) = i + \ell - 1$.

The string $\hat{S}$ is then parsed into blocks. A bijective function $\pi : \Sigma \to [1\,..\,|\Sigma|]$ is chosen uniformly at random; we call it a permutation. We then extend π to $\hat{\Sigma}$ so that $\pi(\boxed{a^\ell}) = \pi(a)$, that is, the value on a metasymbol is inherited from the underlying symbol. Note that no two consecutive symbols in $\hat{S}$ have the same π value. We then define local minima in $\hat{S}$, and these are used to parse $\hat{S}$ (and S) into blocks.

Definition 3.2. Given a string S, its corresponding string $\hat{S}[1\,..\,\hat{n}]$, and a permutation π on the alphabet of S, a local minimum of $\hat{S}$ is defined as any position $\hat{i}$ such that $1 < \hat{i} < \hat{n}$ and $\pi(\hat{S}[\hat{i}-1]) > \pi(\hat{S}[\hat{i}]) < \pi(\hat{S}[\hat{i}+1])$.

Definition 3.3. The parsing of $\hat{S}$ partitions it into a sequence of blocks. The blocks end at position $\hat{n}$ and at every local minimum. The parsing of $\hat{S}$ induces a parsing on S: If a block ends at $\hat{S}[\hat{i}]$, then a block ends at $S[\mathrm{map}^{-\mathrm{L}}(\hat{i})]$.

Note that, by definition, the first block starts at $S[1]$. When applied on texts $S[1\,..\,n]$, it will hold that $\hat{S}[1] = \#$ and $\hat{S}[\hat{n}] = \$$, so $\hat{S}$ will also be a text (i.e., it will have distinct sentinel symbols at the beginning and at the end). Further, we will always force that $\pi(\$) = 1$ and $\pi(\#) = 2$, which guarantees that there cannot be local minima in $\hat{S}[1\,..\,2]$ nor in $\hat{S}[\hat{n}-1\,..\,\hat{n}]$. Together with the fact that there cannot be two consecutive local minima, this yields the following observation.

Observation 3.4. Every block in S or $\hat{S}$ is formed by at least two consecutive elements (symbols or metasymbols, respectively).

Definition 3.5. We say that a position $p < n$ of the parsed text S is a block boundary if a block ends at position p. For every non-empty fragment $S[i\,..\,j]$ of S, we define

$$B(i,j) = \{p - i : i \leq p < j \text{ and } p \text{ is a block boundary}\}.$$

Moreover, for every integer $c \geq 0$, we define subsets $L(i,j,c)$ and $R(i,j,c)$ of $B(i,j)$ consisting of the $\min(c, |B(i,j)|)$ smallest and largest elements of $B(i,j)$, respectively.

Observe that any fragment $S[i\,..\,j]$ intersects a sequence of $1 + |B(i,j)|$ blocks (the first and the last block might not be contained in the fragment). We are interested in locally contracting parsings, where this number of blocks is smaller than the fragment's length by a constant factor.

Definition 3.6. A parsing is locally contracting if there exist constants α and $\beta < 1$ such that $|B(i,j)| \leq \alpha + \beta|S[i\,..\,j]|$ for every fragment $S[i\,..\,j]$ of S.

Lemma 3.7. The parsing of S from Definition 3.3 is locally contracting with $\alpha = 0$ and $\beta = \frac{1}{2}$.

Proof. By Observation 3.4, adjacent positions in S cannot both be block boundaries. Hence, $|B(i,j)| \leq \lceil \frac{j-i}{2} \rceil = \lfloor \frac{j-i+1}{2} \rfloor = \lfloor \frac{1}{2}|S[i\,..\,j]| \rfloor \leq \frac{1}{2}|S[i\,..\,j]|$. □

We formally define locally consistent parsings as follows.

Definition 3.8. A parsing is locally consistent if there exists a constant c_p such that for every pair of matching fragments $S[i\,..\,j] = S[i'\,..\,j']$ it holds that $B(i,j) \setminus B(i',j') \subseteq L(i,j,c_p) \cup R(i,j,c_p)$, that is $B(i,j)$ and $B(i',j')$ differ by at most c_p smallest and c_p largest elements.

Next, we prove local consistency of our parsing.

Lemma 3.9. The parsing of S from Definition 3.3 is locally consistent with $c_p = 1$. More precisely, if $S[i\,..\,j] = S[i'\,..\,j']$ are matching fragments of S, then

$$B(i,j) \setminus \{\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i)) - i\} = B(i',j') \setminus \{\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i)) - i\}.$$

Proof. By definition, a block boundary is a position q such that $q = \mathrm{map}^{-\mathrm{L}}(\hat{q})$ for a local minimum $\hat{S}[\hat{q}]$ in $\hat{S}$. Hence, a position q, with $1 < q < n$, is a block boundary if and only if $\pi(S[q]) < \pi(S[q+1])$ and $\pi(S[q]) < \pi(S[r])$, where $r = \mathrm{map}^{-\mathrm{F}}(\mathrm{map}(q)) - 1$ is the rightmost position to left of q with $S[r] \neq S[q]$.

Consider a position p, with $i < p < j$, and the corresponding position $p' = p - i + i'$. If $p > \mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i))$, then the positions $r = \mathrm{map}^{-\mathrm{F}}(\mathrm{map}(p)) - 1$ and $r' = \mathrm{map}^{-\mathrm{F}}(\mathrm{map}(p')) - 1$ satisfy $r' - i' = r - i \geq 0$. Hence, p is a block boundary if and only if p' is one. On the other hand, if $p < \mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i))$, then neither p nor p' is a block boundary because $S[p] = S[p+1]$ and $S[p'] = S[p'+1]$.

Consequently, only the position $p = \mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i))$ is a block boundary not necessarily if and only if p' is one. That is, $B(i,j) \setminus \{\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i)) - i\} = B(i',j') \setminus \{\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i)) - i\}$. Moreover, since $\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i)) - i$ may only be the leftmost element of $B(i,j)$, this yields $B(i,j) \setminus B(i',j') \subseteq L(i,j,1)$, and therefore the parsing is locally consistent with $c_p = 1$. □

We conclude this section by defining block extensions and proving that they are sufficiently long to ensure that the block is preserved within the occurrences of its extension. This property will be used several times in subsequent sections.

Definition 3.10. Let $S[i\,..\,j]$, with $1 < i < j < n$, be a block in S. The extension of the block $S[i\,..\,j]$ is defined as $S[i^e\,..\,j^e]$, where $i^e = \mathrm{map}^{-\mathrm{F}}(\mathrm{map}(i-1)) - 1$ and $j^e = j + 1$.

Note that the first and last blocks cannot be extended. For the remaining blocks $S[i\,..\,j]$, the definition is sound because $\mathrm{map}(i-1) > 1$ and $j < n$ since $\mathrm{map}(i-1)$ and $\mathrm{map}(j)$ are

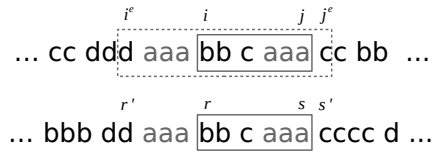

Fig. 1. Illustration of Lemma 3.11. Local minima are shown in gray. Recall $r' = r^e$ and $s' = s^e$.

local minima of S'. Further, note that the block extension spans only the last symbol of the metasymbol $\hat{S}[\mathrm{map}(i^e)]$ and the first of $\hat{S}[\mathrm{map}(j^e)]$.

Lemma 3.11. Let $S[i^e\,..\,j^e]$ be the extension of a block $S[i\,..\,j]$. If $S[r'\,..\,s']$ matches $S[i^e\,..\,j^e]$, then $S[r'\,..\,s']$ contains the same block $S[r\,..\,s] = S[i\,..\,j]$, whose extension is precisely $S[r^e\,..\,s^e] = S[r'\,..\,s']$. Furthermore, $r - r^e = i - i^e$ and $s^e - s = j^e - j$.

Proof. Observe that $\mathrm{map}^{-\mathrm{L}}(\mathrm{map}(i^e)) = i^e$, so Lemma 3.9 yields $B(r', s') \setminus \{0\} = B(i^e, j^e) \setminus \{0\}$. Moreover, $\mathrm{map}(i^e) = \mathrm{map}(i-1) - 1$ and $\mathrm{map}(j^e) = \mathrm{map}(j) + 1$, so $B(i^e, j^e) = \{i-1-i^e, j-i^e\}$ due to Observation 3.4. Hence, $B(r', s')\setminus\{0\} = \{i-1-i^e, j-i^e\}$, and therefore $S[r\,..\,s]$ is a block, where $r = i - i^e + r'$ and $s = j - i^e + r'$. Figure 1 gives an example.

To complete the proof, notice that $s^e = s + 1 = s'$ and $r^e = \mathrm{map}^{-\mathrm{L}}(\mathrm{map}(r-1)) - 1 = r'$ follows from the fact that $S[r'\,..\,s']$ and $S[i^e\,..\,j^e]$ match. □

4. GRAMMARS WITH LOCALITY PROPERTIES

Consider a context-free grammar (CFG) that generates a string S and only S [Kieffer and Yang 2000]. Each nonterminal must be the left-hand side in exactly one production, and the size g of the grammar is the sum of the right-hand sides of the productions. It is NP-complete to compute the smallest grammar for a string S [Charikar et al. 2005], but it is possible to build grammars of size $g = O(z \log(|S|/z))$ if the Lempel–Ziv parsing of S consists of z phrases [Gawrychowski 2011, Lemma 8].[2]

If we allow, in addition, rules of the form $A \to A_1^s$, where $s \geq 2$, taken to be of size 2 for technical convenience, the result is a run-length context-free grammar (RLCFG) [Nishimoto et al. 2016]. These grammars encompass CFGs and are intrinsically more powerful; for example, the smallest CFG for the string family $S = a^n$ is of size $\Theta(\log n)$ whereas already an RLCFG of size $O(1)$ can generate it.

The parse tree of a CFG has internal nodes labeled with nonterminals and leaves labeled with terminals. The root is the initial symbol and the concatenation of the leaves yields S: the ith leaf is labeled $S[i]$. If $A \to A_1 \cdots A_s$, then any node labeled A has s children, labeled $A_1, \ldots, A_s$. In the parse tree of a RLCFG, rules $A \to A_1^s$ are represented as a node labeled A with s children nodes labeled A_1. The following definition describes the substring of S generated by each node.

Definition 4.1. If the leaves descending from a parse tree node v are the ith to the jth leaves, we say that v generates $S[i\,..\,j]$ and that v is projected to the interval $proj(v) = [i\,..\,j]$.

The subtrees of equally labeled nodes are identical and generate the same strings, so we speak of the strings generated by the grammar symbols. We call $\exp(A)$ the expansion of nonterminal A, that is, the string it generates (or the concatenation of the leaves under

[2]There are older constructions [Rytter 2003; Charikar et al. 2005], but they refer to a restricted Lempel–Ziv variant where sources and phrases cannot overlap.

any node labeled A in the parse tree), and $|A| = |\exp(A)|$. For terminals a, we assume $\exp(a) = a$.

A grammar is said to be balanced if the parse tree is of height $O(\log n)$. A stricter concept is the following one.

Definition 4.2. A grammar is locally balanced if there exists a constant b such that, for any nonterminal A, the height of any parse tree node labeled A is at most $b \cdot \log |A|$.

4.1. From parsings to balanced grammars

We build an RLCFG on a text $T[1\,..\,n]$ using our parsing of Section 3. In the first pass, we collect the distinct runs a^ℓ with $\ell \geq 2$ and create run-length nonterminals of the form $A \to a^\ell$ to replace the corresponding runs in T. The resulting sequence is analogous to $\hat{T}$, where a nonterminal $A \to a^\ell$ stands for the metasymbol $\boxed{a^\ell}$, and the terminal a stands for the metasymbol $\boxed{a^1}$.

Next, we choose a permutation π and perform a pass on the new text $\hat{T}$, defining the blocks based on local minima according to Definition 3.3. Each distinct block $A_1 \cdots A_k$ is replaced by a distinct nonterminal A with the rule $A \to A_1 \cdots A_k$ (each A_i can be a symbol of Σ or a run-length nonterminal created in the first pass). The blocks are then replaced by those created nonterminals A, which results in a string T'. The string T' is of length $n' \leq \lfloor n/2 \rfloor$, by Observation 3.4. Note that the first and last symbols of T' expand to blocks that contain # and \$, respectively, and thus they are unique too. We can then regard T' as a text, by having its first nonterminal, $T'[1]$, play the role of #, and the last, $T'[n']$, play the role of \$.

The process is then repeated again on T', and iterated for $h \leq \lfloor \log n \rfloor$ rounds, until a single nonterminal is obtained. This is the initial symbol of the grammar. We denote by $T_r[1\,..\,n_r]$ the text created in round r, so $T_0 = T$ and $T_1 = T'$. We also denote by $\hat{T}_r[1\,..\,\hat{n}_r]$ the intermediate text obtained by collapsing runs in T_r. Figure 2 exemplifies the grammars we build and the corresponding parse tree.

The height of the grammar is at most $2h \leq 2\lfloor \log n \rfloor$, because we create run-length rules and then block-rules in each round. This grammar is then balanced because, by Observation 3.4, $n_r \leq n/2^r$. Moreover, the grammar is locally balanced.

Lemma 4.3. The grammar we build from our parsing is locally balanced with $b = 2$.

Proof. Because of Observation 3.4, any subtree rooted at a nonterminal A in the parse tree (at least) doubles the number of nodes per round towards the leaves. If A is formed in round r, then the subtree has height at most $2r$, and the expansion satisfies $|A| \geq 2^r$. The height of the subtree rooted at A is thus at most $2r \leq 2 \log |A|$. □

4.2. Local consistency properties

We now formalize the concept of local consistency for our grammars. For each $r \in [0\,..\,h]$, the subsequent characters of T_r naturally correspond to nodes of the parse tree of T, and the fragments $T[i\,..\,j]$ generated by these nodes form a decomposition of T. We denote this parsing of T by $\mathcal{P}_r$. In other words, $T[i\,..\,j]$ is a block of $\mathcal{P}_r$ if and only if $[i\,..\,j] = proj(v)$ for some node v labeled by a symbol in T_r. We refer to the blocks and block boundaries in this parsing as level-r blocks and level-r block boundaries. Analogously, we define a parsing $\hat{\mathcal{P}}_r$ with blocks corresponding to subsequent symbols of $\hat{T}_r$, and we refer to the underlying blocks and block boundaries as level-r runs and level-r run boundaries; see Figure 2.

Note that every level-r run boundary is also a level-r block boundary, and every level-$(r+1)$ block boundary is also a level-r run boundary. Moreover, by Observation 3.4, at most one out of every two subsequent level-r run boundaries can be a level-$(r+1)$ block boundary.

Permutations

$\pi(\$_3) = 1$
$\pi(\#_3) = 2$
$\pi(\$_2) = 1$
$\pi(\#_2) = 2$
$\pi(\mathsf{E}) = 3$
$\pi(\mathsf{D}) = 4$

$\pi(\$_1) = 1$
$\pi(\#_1) = 2$
$\pi(\mathsf{A}) = 3$
$\pi(\mathsf{C}) = 4$
$\pi(\mathsf{B}) = 5$

$\pi(\$) = 1$
$\pi(\#) = 2$
$\pi(\mathsf{d}) = 3$
$\pi(\mathsf{c}) = 4$
$\pi(\mathsf{b}) = 5$
$\pi(\mathsf{a}) = 6$

Grammar

$\mathsf{S} \to \#_3\$_3$
$\$_3 \to \mathsf{DD}\$_2$
$\#_3 \to \#_2\mathsf{DDE}$
$\$_2 \to \mathsf{B}\$_1$
$\#_2 \to \#_1\mathsf{BAA}$
$\mathsf{E} \to \mathsf{BBAA}$
$\mathsf{D} \to \mathsf{BC}$
$\$_1 \to \mathsf{c}\$$
$\#_1 \to \#\mathsf{ad}$
$\mathsf{A} \to \mathsf{aac}$
$\mathsf{C} \to \mathsf{aabc}$
$\mathsf{B} \to \mathsf{bd}$

Parse tree

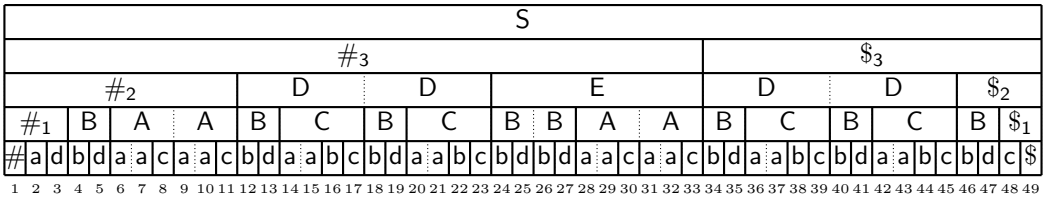

Fig. 2. An example of the construction of our grammar. The top-left part shows the permutations π assigned in each level, and the top-right part gives the complete grammar built (for simplicity we omit run-length nonterminals). The parse tree, shown on the bottom, also omits run-length nonterminals. The texts T_r correspond to the subsequent levels of the parse tree (starting from the bottom). Level-r block boundaries that are not run boundaries are depicted using dotted lines. For example, $T_2 = \#_2\mathsf{DDEDD}\$_2$, $n_2 = 7$, and the level-2 block boundaries are $11, 17, 23, 33, 39, 45$. On the other hand, $\hat{n}_2 = 5$ and the level-2 run boundaries are $11, 23, 33, 45$. The corresponding parsings $\mathcal{P}_2$ and $\hat{\mathcal{P}}_2$ decompose T as $\#\mathsf{adbdaacaac|bdaabc|bdaabc|bdbdaacaac|bdaabc|bdaabc|bdc}\$$ and $\#\mathsf{adbdaacaac|bdaabcbdaabc|bdbdaacaac|bdaabcbdaabc|bdc}\$$, respectively.

Definition 4.4. For every non-empty fragment $T[i\,..\,j]$ of T, the sets defined according to Definition 3.5 for the parsing $\mathcal{P}_r$ are denoted $B_r(i,j)$, $L_r(i,j,c)$, and $R_r(i,j,c)$. Analogously, we denote by $\hat{B}_r(i,j)$, $\hat{L}_r(i,j,c)$, and $\hat{R}_r(i,j,c)$ the sets defined for the parsing $\hat{\mathcal{P}}_r$.

These notions let us reformulate Lemma 3.9 so that it is directly applicable at every level r.

Lemma 4.5. If matching fragments $T[i\,..\,j]$ and $T[i'\,..\,j']$ both consist of full level-r blocks, then the corresponding fragments of T_r also match, so $B_r(i,j) = B_r(i',j')$ and $\hat{B}_r(i,j) = \hat{B}_r(i',j')$. Moreover, $B_{r+1}(i,j) \setminus \{\min \hat{B}_r(i,j)\} = B_{r+1}(i',j') \setminus \{\min \hat{B}_r(i,j)\}$ if $\hat{B}_r(i,j) \neq \emptyset$, and $B_{r+1}(i,j) = B_{r+1}(i',j') = \emptyset$ otherwise.

Proof. We proceed by induction on r. The first two claims hold trivially for $r = 0$: the fragments $T[i\,..\,j]$ and $T[i'\,..\,j']$ of $T_0 = T$ clearly match, and $B_0(i,j) = [0\,..\,j-i-1] = B_0(i',j')$. For $r > 0$, on the other hand, $T[i\,..\,j]$ and $T[i'\,..\,j']$ consist of full level-$(r-1)$ blocks, so the inductive assumption yields that the corresponding fragments of T_{r-1} also match and that $B_r(i,j) = B_r(i',j') = \emptyset$ or $B_r(i,j) \setminus \{\min \hat{B}_{r-1}(i,j)\} = B_r(i',j') \setminus \{\min \hat{B}_{r-1}(i,j)\}$. In the latter case, we observe that $i-1$ and $i+\min \hat{B}_{r-1}(i,j)$ are subsequent level-$(r-1)$ run boundaries while $i-1$ is a level-r block boundary, or $i = 1$ and $i + \min \hat{B}_{r-1}(i,j)$ is the leftmost level-$(r-1)$ run boundary. Either way, $i + \min \hat{B}_{r-1}(i,j)$ cannot be a level-r block boundary due to Observation 3.4, so $B_r(i,j) \setminus \{\min \hat{B}_{r-1}(i,j)\} = B_r(i,j)$. A symmetric argument proves that $B_r(i',j') \setminus \{\min \hat{B}_{r-1}(i,j)\} = B_r(i',j')$, which lets us conclude that $B_r(i,j) = B_r(i',j')$. Hence, the matching fragments of T_{r-1} corresponding to $T[i\,..\,j]$ and $T[i'\,..\,j']$ are parsed into the same blocks so the corresponding fragments of T_r also match.

To prove the other two claims for arbitrary $r \geq 0$, notice that the fragments of T_r corresponding to $T[i\,..\,j]$ and $T[i'\,..\,j']$ are occurrences of the same string, denoted P_r. Hence, $\hat{B}_r(i,j)$ and $\hat{B}_r(i',j')$ are equal as they both correspond to the run boundaries in P_r. If P_r consists of a single run (i.e., if $\hat{B}_r(i,j) = \emptyset$), then clearly $B_{r+1}(i,j) = B_{r+1}(i',j') = \emptyset$. Otherwise, Lemma 3.9 implies $B_{r+1}(i,j) \setminus \{\min \hat{B}_r(i,j)\} = B_{r+1}(i',j') \setminus \{\min \hat{B}_r(i,j)\}$. □

Nevertheless, we define local consistency of a grammar as a stronger property than the one expressed in Lemma 4.5: we require that $B_r(i,j)$ and $B_r(i',j')$ resemble each other even if the matching fragments $T[i\,..\,j]$ and $T[i'\,..\,j']$ do not consist of full blocks.

Definition 4.6. The grammar we build is locally consistent if there is a constant c_g such that the parsings $\mathcal{P}_r$ are all locally consistent with constant c_g.

In the rest of this section, we prove that our grammar is locally consistent with constant $c_g = 3$. Our main tool is the following construction of sets $B_r(P)$ and $\hat{B}_r(P)$, consisting of the positions (relative to P) of context-insensitive level-r block and run boundaries that are common to all occurrences of P in T. Despite these sets being defined based on an occurrence of P in T, we show in Lemma 4.10 that they do not depend on the choice of the occurrence.

Definition 4.7. Let P be a substring of T and let $T[i\,..\,j]$ be its arbitrary occurrence in T. The sets $B_r(P)$ and $\hat{B}_r(P)$ for $r \geq 0$ are defined recursively, with $X + \delta = \{x + \delta : x \in X\}$.

$$B_r(P) = \begin{cases} [0\,..\,|P|-2] & \text{if } r = 0, \\ B_r(i+1+\min \hat{B}_{r-1}(P), i+\max B_{r-1}(P)) + 1 + \min \hat{B}_{r-1}(P) & \text{if } \hat{B}_{r-1}(P) \neq \emptyset, \\ \emptyset & \text{if } \hat{B}_{r-1}(P) = \emptyset; \end{cases}$$

$$\hat{B}_r(P) = \begin{cases} \hat{B}_r(i+1+\min B_r(P), i+\max B_r(P)) + 1 + \min B_r(P) & \text{if } B_r(P) \neq \emptyset, \\ \emptyset & \text{if } B_r(P) = \emptyset. \end{cases}$$

Our index also relies on a set $M(P)$ designed as a superset of $B_r(i,j) \setminus B_r(P)$ for every r and every occurrence $T[i\,..\,j]$ of P. In other words, $M(P)$ contains, for each r, positions within P that may be level-r block boundaries in some but not necessarily all occurrences of P.

Definition 4.8. For a substring P of T, the set $M(P)$ is defined to contain $\min B_r(P)$ and $\max B_r(P)$ for every $r \geq 0$ with $B_r(P) \neq \emptyset$, and $\min \hat{B}_r(P)$ for every $r \geq 0$ with $\hat{B}_r(P) \neq \emptyset$.

Example 4.9. Consider $P = \mathsf{dbdaacaacbdaabcbdaabcbd}$ with occurrences $T[3\,..\,25]$ and $T[25\,..\,47]$ in the text T of Figure 2. For $r = 0$, we define $B_0(P) = [0\,..\,21]$ and set $\hat{B}_0(P) = \{1,2,4,5,7,8,9,10,12,13,14,15,16,18,19,20\} = 1 + \hat{B}_0(4,24) = 1 + \hat{B}_0(26,46)$. For $r = 1$, we set $B_1(P) = \{2,5,8,10,14,16,20\} = 2 + B_1(5,24) = 2 + B_1(27,46)$ and $\hat{B}_1(P) = \{8,10,14,16\} = 3 + \hat{B}_1(6,23) = 3 + \hat{B}_1(28,45)$. For $r = 2$, we set $B_2(P) = \{14\} = 9 + B_2(12,23) = 9 + B_2(34,45)$ and $\hat{B}_2(P) = \emptyset = 15 + \hat{B}_2(18,17) = 15 + \hat{B}_2(40,39)$. For $r \geq 3$, we have $B_r(P) = \hat{B}_r(P) = \emptyset$. Consequently, $M(P) = \{0,1,2,8,14,20,21\}$; see Figure 3.

We now show $B_r(P)$ contains all the level-r block boundaries in any occurrence of P in T except possibly the first 3 and the last one, but those missing boundaries belong to $M(P)$.

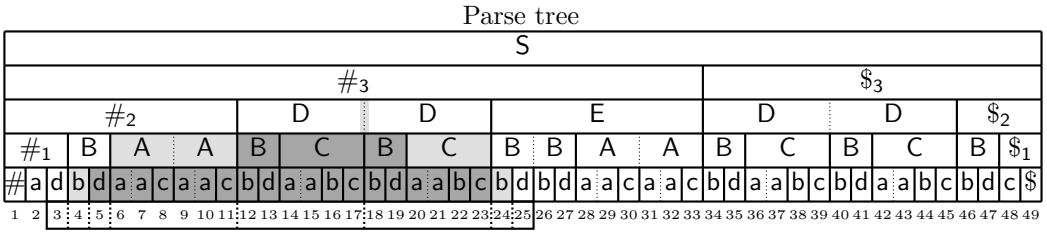

Fig. 3. The construction of $M(P)$ for the pattern P analyzed in Example 4.9, illustrated on the occurrence of P at $T[3\,..\,25]$. In light gray, we show the area between $\min B_r(P)+1$ and $\max B_r(P)$, and in dark gray the area between $\min \hat{B}_r(P)+1$ and $\max \hat{B}_r(P)$. At each level r, the parsing between those extremes is always the same for every occurrence of P; see $T[25\,..\,47]$ for example. The set $M(P)$ contains the relative position of $\min B_r(P)$, $\max B_r(P)$, and $\min \hat{B}_r(P)$ for every r, marked by dotted lines between indices.

Lemma 4.10. For every substring P of T and every $r \geq 0$, the sets $B_r(P)$ and $\hat{B}_r(P)$ do not depend on the choice of an occurrence $T[i\,..\,j]$ of P. Moreover,

$$B_r(P) \cup L_r(i,j,3) \cup R_r(i,j,1) = B_r(i,j) \subseteq B_r(P) \cup M(P). \tag{1}$$

Proof. We proceed by induction on r, proving the independence of $\hat{B}_r(P)$ only at step $r+1$. In the base case, $B_0(P) = [0\,..\,|P|-2]$ does not depend on the choice of the occurrence, and Eq. (1) is satisfied because $B_0(P) = B_0(i,j)$.

For the inductive step, we assume the claims hold for $B_r(P)$. If $B_r(P) = \emptyset$, then $\hat{B}_r(P) = B_{r+1}(P) = \emptyset$ do not depend on the occurrence of P. The inductive assumption yields $B_{r+1}(i,j) \subseteq B_r(i,j) \subseteq M(P) = B_{r+1}(P) \cup M(P)$ and $|B_{r+1}(i,j)| \leq |B_r(i,j)| = |L_r(i,j,3) \cup R_r(i,j,1)| \leq 4$, so $L_{r+1}(i,j,3) \cup R_{r+1}(i,j,1) = B_{r+1}(i,j)$ and Eq. (1) is satisfied.

We henceforth assume that $B_r(P) \neq \emptyset$. Since $B_r(P) \subseteq B_r(i,j)$, both $i+\min B_r(P)$ and $i+\max B_r(P)$ are level-r block boundaries, and therefore $T[i+\min B_r(P)+1\,..\,i+\max B_r(P)]$ consists of full level-r blocks. We conclude from Lemma 4.5 that $\hat{B}_r(P)$, as defined in Definition 4.7, does not depend on the occurrence of P. Moreover, the only position between $i+\min B_r(P)$ and $i+\max B_r(P)$ that may or may not be a level-$(r+1)$ block boundary depending on the context of $T[i\,..\,j]$ is $i+\min \hat{B}_r(P)$ provided that $\hat{B}_r(P) \neq \emptyset$. In particular, $B_{r+1}(P)$, as defined in Definition 4.7, also does not depend on the occurrence of P.

To prove that $B_{r+1}(P)$ satisfies Eq. (1), we consider two cases. First, suppose that $\hat{B}_r(P) = \emptyset$, that is, there are no level-r run boundaries between $i+\min B_r(P)$ and $i+\max B_r(P)$. Since $\hat{B}_r(i,j) \subseteq B_r(i,j)$, the inductive assumption $B_r(i,j) = B_r(P) \cup L_r(i,j,3) \cup R_r(i,j,1)$ implies $\hat{B}_r(i,j) \subseteq \{\min B_r(P), \max B_r(P)\} \cup L_r(i,j,3) \cup R_r(i,j,1)$, while $B_r(i,j) \subseteq B_r(P) \cup M(P)$ yields $\hat{B}_r(i,j) \subseteq \{\min B_r(P), \max B_r(P)\} \cup M(P) = M(P)$, where the equality follows from Definition 4.8. The former assertions yields $|\hat{B}_r(i,j)| \leq 6$, and since $B_{r+1}(i,j) \subseteq \hat{B}_r(i,j)$ cannot contain two consecutive elements of $\hat{B}_r(i,j)$ by Observation 3.4, we conclude that $|B_{r+1}(i,j)| \leq 3$. In particular, since $B_{r+1}(P) = \emptyset$ according to Definition 4.7, we have $B_{r+1}(P) \cup L_{r+1}(i,j,3) \cup R_{r+1}(i,j,1) = B_{r+1}(i,j) \subseteq B_{r+1}(P) \cup M(P)$ as claimed.

Next, suppose that $\hat{B}_r(P) \neq \emptyset$. Definition 4.7 clearly implies $B_{r+1}(P) \subseteq B_{r+1}(i,j)$, so it remains to prove that $B_{r+1}(i,j)$ is a subset of both $B_{r+1}(P) \cup L_{r+1}(i,j,3) \cup R_{r+1}(i,j,1)$ and $B_{r+1}(P) \cup M(P)$. We take $q \in B_{r+1}(i,j)$ and consider three cases.

(1) If $q \le \min \hat{B}_r(P)$, then $q \in (L_r(i,j,3) \cap M(P)) \cup \{\min B_r(P), \min \hat{B}_r(P)\}$ and therefore $q \in \hat{L}_r(i,j,5)$.[3] Since $B_{r+1}(i,j)$ cannot contain two consecutive elements of $\hat{B}_r(i,j)$ due to Observation 3.4, $q \in B_{r+1}(i,j) \cap \hat{L}_r(i,j,5)$ implies $q \in L_{r+1}(i,j,3)$. Finally, $q \in M(P) \cup \{\min B_r(P), \min \hat{B}_r(P)\} = M(P)$, where the equality holds due to Definition 4.8.
(2) If $q \ge \max B_r(P)$, then $q \in (R_r(i,j,1) \cap M(P)) \cup \{\max B_r(P)\}$ and therefore $q \in \hat{R}_r(i,j,2)$.[4] Since $B_{r+1}(i,j)$ cannot contain two consecutive elements of $\hat{B}_r(i,j)$ due to Observation 3.4, $q \in B_{r+1}(i,j) \cap R_r(i,j,2)$ implies $q \in R_{r+1}(i,j,1)$. Finally, $q \in M(P) \cup \{\max B_r(P)\} = M(P)$, where the equality holds due to Definition 4.8.
(3) If $\min \hat{B}_r(P) < q < \max B_r(P)$, then $q + i$ is a level-$(r+1)$ block boundary and $q \in B_{r+1}(P)$ by Definition 4.7. □

Lemma 4.10 implies that the grammar constructed in this section is locally consistent with $c_g = 3$. We conclude this section with a further characterization of the set $M(P)$.

Lemma 4.11. For each substring $P = T[i\,..\,j]$, the set $M(P)$ satisfies the following properties:

(1) If $B_r(i,j) \neq \emptyset$ for some $r \ge 0$, then $\min B_r(i,j) \in M(P)$,
(2) If $\hat{B}_r(i,j) \neq \emptyset$ for some $r \ge 0$, then $\min \hat{B}_r(i,j) \in M(P)$,
(3) $|M(P)| \le 3\lceil \log |P| \rceil$.

Proof. To prove (1) for any r, note that $B_r(P) \subseteq B_r(i,j) \subseteq B_r(P) \cup M(P)$ by Lemma 4.10. If $\min B_r(i,j) \notin B_r(P)$, then it belongs to $M(P)$. Otherwise, it must be equal to $\min B_r(P)$, which is in $M(P)$ by Definition 4.8.

The proof of (2) is similar: Since $\hat{B}_r(i,j) \subseteq B_r(i,j)$, either $\min \hat{B}_r(i,j) \in B_r(i,j) \setminus B_r(P) \subseteq M(P)$, or $\min \hat{B}_r(i,j) \in B_r(P)$. If $\min \hat{B}_r(i,j) \in \{\min B_r(P), \max B_r(P)\}$, then it is in $M(P)$ by Definition 4.8. Otherwise, $\min B_r(P) < \min \hat{B}_r(i,j) < \max B_r(P)$ and, by the choice of $\hat{B}_r(P)$ in Definition 4.7, $\min \hat{B}_r(i,j) = \min \hat{B}_r(P)$ is also in $M(P)$ by Definition 4.8.

To prove (3), notice that $|B_r(P)| \le \frac{1}{2}|B_{r-1}(P)|$ holds for all r due to Definition 4.7, Observation 3.4, and $\min B_{r-1}(P) \notin B_r(P)$. This implies $|B_r(P)| \le |B_0(P)| \cdot 2^{-r} < |P| \cdot 2^{-r}$, and therefore $\hat{B}_r(P) = B_r(P) = \emptyset$ for $r \ge \log |P|$. Definition 4.8 now yields the claim. □

5. BOUNDING OUR GRAMMAR IN TERMS OF ATTRACTORS

Let us first define the concept of attractors in a string [Kempa and Prezza 2018].

Definition 5.1 ([Kempa and Prezza 2018]). An attractor of a string S is a set $\Gamma \subseteq [1\,..\,n]$ of positions in S such that each non-empty substring Q of S has an occurrence $S[i\,..\,j]$ containing an attractor position, i.e., satisfying $i \le p \le j$ for some $p \in \Gamma$.

In this section, we show that the RLCFG of Section 4 is of size $g = O(\gamma \log(n/\gamma))$, where γ is the size of a smallest attractor of T. The key is to prove that distinct nonterminals are formed only around the attractor elements. For this, we first prove that $T'[1\,..\,n']$, where the blocks of T are converted into nonterminals, contains an attractor of size at most 3γ.

[3] By the choice of $\hat{B}_r(P)$ in Definition 4.7, there are no level-r run boundaries between $\min B_r(P)$ and $\min \hat{B}_r(P)$. Note that $q < \min B_r(P)$ yields $q \notin B_r(P)$. Since $q \in B_r(i,j)$, by the inductive assumption $q \notin B_r(P)$ implies $q \in L_r(i,j,3) \cap M(P)$ ($q \notin R_r(i,j,1)$ because $q < \min B_r(P) \in B_r(i,j)$). For the same reason, $L_r(i,j,3) \cup \{\min B_r(P)\} \subseteq L_r(i,j,4)$ and $\min \hat{B}_r(P) \in \hat{L}_r(i,j,5)$.

[4] Note that $q > \max B_r(P)$ yields $q \notin B_r(P)$. Since $q \in B_r(i,j)$, by the inductive assumption $q \notin B_r(P)$ implies $q \in R_r(i,j,1) \cap M(P)$ ($q \notin L_r(i,j,3)$ because $q > \max B_r(P) > \min \hat{B}_r(P) > \min B_r(P)$ and these 3 elements belong to $B_r(i,j)$). For the same reason, $R_r(i,j,1) \cup \{\max B_r(P)\} \subseteq R_r(i,j,2)$.

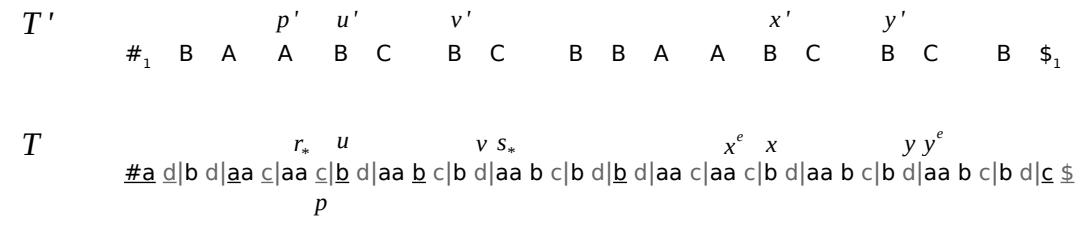

Fig. 4. Illustration of Lemma 5.2. We underlined positions in T corresponding to a particular string attractor.

Lemma 5.2. Let Γ be an attractor of T, and let $\Gamma' = \bigcup_{p\in\Gamma}[p'-1\,..\,p'+1]$, where p' is the position in T' of the nonterminal that covers p in T. Then Γ' is an attractor of T'.

Proof. Figure 4 illustrates the proof. Consider an arbitrary substring $T'[x'\,..\,y']$, with $x' \geq 3$ and $y' \leq n'-2$; otherwise the substring crosses an attractor because 1 and n are in Γ. This is a sequence of consecutive nonterminals, each corresponding to a block in T. Let $T[x\,..\,y]$ be the substring of T formed by all the blocks that map to $T'[x'\,..\,y']$. The union of their extensions is also a substring $T[x^e\,..\,y^e]$ of T. Since Γ is an attractor in T, there exists a copy $T[r_*\,..\,s_*] = T[x^e\,..\,y^e]$ that includes an element $p \in \Gamma$, $r_* \leq p \leq s_*$.

Consider any block $T[i\,..\,j]$ inside $T[x\,..\,y]$. Its extension $T[i^e\,..\,j^e]$ is contained in $T[x^e\,..\,y^e]$, so a copy $T[r'\,..\,s']$ of $T[i^e\,..\,j^e]$ appears inside $T[r_*\,..\,s_*]$. By Lemma 3.11, the block $T[i\,..\,j]$ also forms a block $T[r\,..\,s]$ inside $T[r'\,..\,s']$, at the same relative position; furthermore, $T[r'\,..\,s'] = T[r^e\,..\,s^e]$ is the extension of $T[r\,..\,s]$.

Since this happens for every block $T[i\,..\,j]$ inside $T[x\,..\,y]$, which is a sequence of blocks, it follows that $T[x\,..\,y]$ appears inside $T[r_*\,..\,s_*]$, as a subsequence $T[u\,..\,v]$ of blocks; furthermore, its extension $T[u^e\,..\,v^e]$ coincides with $T[r_*\,..\,s_*]$ and thus contains p. Moreover, $T[u\,..\,v]$ maps to a substring $T'[u'\,..\,v'] = T'[x'\,..\,y']$.

Since $v^e = v+1$ and $u^e = \mathrm{map}^{-\mathrm{F}}(\mathrm{map}(u-1))-1$, due to Observation 3.4, the fragments $T[u^e\,..\,u-1]$ and $T[v+1\,..\,v^e]$ are contained within single blocks. Therefore, the position p' to which p is mapped in T' belongs to $T'[u'-1\,..\,v'+1]$. Consequently, $T'[u'\,..\,v']$ contains a position in Γ'. □

We now show that the first round contributes $O(\gamma)$ to the size of the final RLCFG. In this bound, we only count the the sizes of the generated rules; the whole accounting will be done in Theorem 5.4. The idea is to show that the 3 distinct blocks formed around each attractor element have expected length $O(1)$.

Lemma 5.3. The first round of parsing contributes $O(\gamma)$ to the grammar size, in expectation. Further, a parsing producing a grammar of size $O(\gamma)$ is found in $O(n)$ expected time provided that γ is known.

Proof. Let us first focus on block-forming rules; we consider the run-length rules in the next paragraph. The right-hand sides of the block-forming rules correspond to the distinct blocks formed in $\hat{T}$, that is, to single symbols in T'. All the distinct symbols in T', in turn, appear at positions of Γ'. By Theorem 5.2, Γ' is of size at most 3γ; therefore, there are at most 3γ distinct blocks in $\hat{T}$ and in T (i.e., those containing attractor elements of T and their neighboring blocks), and thus at most 3γ distinct nonterminals are formed in the grammar.

We must also show, however, that the sum of the sizes of the right-hand sides of those 3γ productions also add up to $O(\gamma)$. Consider a block of $\hat{T}$ of length ℓ. The right-hand side of its corresponding production is ℓ. Each element of $\hat{T}$ can be a metasymbol, however, so the grammar may indeed include ℓ further run-length nonterminals, contributing up to 2ℓ

to the grammar size. Therefore, each distinct block of length ℓ in $\hat{T}$ contributes at most 3ℓ to the grammar size. We now show that $\ell = O(1)$ in expectation for the 3γ blocks specified above.

Consider an attractor element p and its position $\hat{p} = \text{map}(p)$ when mapped to $\hat{T}$. Let $T[i\,..\,j]$ be the block containing p and let $T[i'..j']$ be its concatenation with the adjacent blocks ($T[i'\,..\,i-1]$ and $T[j+1\,..\,j']$). Moreover, let $\hat{T}[\hat{i}\,..\,\hat{j}]$ and $\hat{T}[\hat{i}'\,..\,\hat{j}']$ be the corresponding fragments of $\hat{T}$, with $\hat{i} = \text{map}(i)$, $\hat{j} = \text{map}(j)$, $\hat{i}' = \text{map}(i')$, and $\hat{j}' = \text{map}(j)$.

Let $\ell^+ = \hat{j}' - \hat{p} + 1$, $\ell^- = \hat{p} - \hat{i}'$, and $\ell = \ell^+ + \ell^-$. Then, 3ℓ is the maximum possible contribution of attractor element p to the grammar size via nonterminals that represent these blocks.

The area $\hat{T}[\hat{p}\,..\,\hat{j}']$ contains at most 2 local minima, at $\hat{j}$ and $\hat{j}'$ (unless $\hat{j}' = \hat{n}$). Note that, between two consecutive local minima, we have a sequence of nondecreasing values of π and then a sequence of nonincreasing values of π. Our area can be covered by 2 such ranges. Hence, if we split the substring of length ℓ^+ into 4 equal parts of length $\ell^+/4$, at least one of them must be monotone (i.e., nondecreasing or nonincreasing) with respect to π.

Note that consecutive symbols in $\hat{T}$ are always different. Further, if there are repeated symbols in a length-d substring of $\hat{T}$, then it cannot be monotone with respect to π. If all the symbols are different, instead, exactly one out of $d!$ permutations π will make the substring increasing and one out of $d!$ will make it decreasing, where d is the length of the substring.

As a result, at most 2 out of $(\ell^+/4)!$ permutations can make one of our length-$(\ell^+/4)$ substrings monotone. If we choose permutations π uniformly at random, then the probability that at least one of our 4 substrings is monotone is at most $8/(\ell^+/4)!$. Since this upper-bounds the probability that $\hat{j}' \geq \hat{p} + \ell^+$, the expected value of ℓ^+ is $O(1)$.[5]

An analogous argument holds for ℓ^- since $\hat{T}[\hat{i}'\,..\,\hat{p}-1]$ can also be covered by at most 2 ranges between consecutive local minima. Adding the expectations of the contributions 3ℓ over the γ attractor elements, we obtain $O(\gamma)$.

If the expectation is of the form $c \cdot \gamma$, then at least half of the permutations produce a grammar of size at most $2c \cdot \gamma$, and thus a Las Vegas algorithm finds a permutation producing a grammar of size at most $2c \cdot \gamma$ after $O(1)$ attempts in expectation. Since at each attempt we parse $T[1\,..\,n]$ in time $O(n)$, we find a suitable permutation in $O(n)$ expected time provided we know γ. □

We now perform $O(\log(n/\gamma))$ rounds of locally-consistent parsing, where the output T' of each round is the input to the next. The length of the string halves in each iteration, and the grammar grows only by $O(\gamma)$ in each round.

Theorem 5.4. Let $T[1\,..\,n]$ have an attractor of size γ. Then there exists a locally-balanced locally-consistent RLCFG of size $g = O(\gamma \log(n/\gamma))$ and height $O(\log(n/\gamma))$ that generates (only) T, and it can be built in $O(n)$ expected time and $O(g)$ working space if γ is known.

Proof. We apply the grammar construction described in Section 4.1, which by Lemmas 4.3 and 4.10, is locally balanced and locally consistent.

We first show that we can build an attractor Γ_r for each T_r formed by γ runs of $m_r \in O(1)$ consecutive positions. This is clearly true for T_0, with $m_0 = 1$. Now assume this holds for T_r. When parsing T_r into blocks to form T_{r+1}, each run of m_r consecutive attractor positions is parsed into at most $1 + \lfloor m_r/2 \rfloor$ consecutive symbols p' in T_{r+1}, as seen in the proof of Lemma 3.7. Lemma 5.2 then shows that, if we expand each such mapped attractor positions p' to $[p'-1\,..\,p'+1]$, we obtain an attractor Γ_{r+1} for T_{r+1}. The union of the expansions

[5] Because $\sum_{k\geq 1} 1/k! = e - 1$.

of $1 + \lfloor m_r/2 \rfloor$ consecutive positions p' creates a run of length $m_{r+1} = 3 + \lfloor m_r/2 \rfloor$. It then holds that Γ_{r+1} is formed by γ runs of at most m_{r+1} positions.

The sequence of values m_r stabilizes. If we solve $m = 3 + \lfloor m/2 \rfloor$, we obtain $\lceil m/2 \rceil = 3$. This solves for $m = 5$ or $m = 6$. Indeed, the value is 5 and is reached soon: $m_0 = 1$, $m_1 = 3$, $m_2 = 4$, $m_3 = 5$, $m_4 = 5$. Therefore, we safely use $m_r \leq 5$ in the following.

The only distinct blocks in each T_r are those forming Γ_{r+1}. Therefore, the parsing of each text T_r produces at most 5γ distinct nonterminal symbols. By Theorem 5.3, we can find in $O(n_r)$ expected time a permutation π_r such that the contribution of the rth round to the grammar size is $O(|\Gamma_r|) = O(\gamma)$.

The sum of the lengths of all T_rs is at most $2n$, thus the total expected construction cost is $O(n)$. We stop after $r^* = \log(n/\gamma)$ rounds. By then, T_{r^*} is of length at most γ and the cumulative size of the grammar is $O(\gamma \cdot r^*) = O(\gamma \log(n/\gamma))$. We add a final rule $S \to T_{r^*}$, which adds γ to the grammar size. The height of the grammar is $O(r^*) = O(\log(n/\gamma))$.

As for the working space, at each new round r we generate a permutation π_r of $|\Sigma_r|$ cells. Since the alphabet size is a lower bound to the attractor size, it holds that $|\Sigma_r| \leq 5\gamma$. We store the distinct blocks that arise during the parsing in a hash table. These are at most 5γ as well, and thus a hash table of size $O(\gamma)$ is sufficient. The rules themselves, which grow by $O(\gamma)$ in each round, add up to $O(g)$ total space. □

5.1. Building the grammar without an attractor

Since finding the size γ of the smallest attractor is NP-complete [Kempa and Prezza 2018], it is interesting that we can find a RLCFG similar to that of Theorem 5.4 without having to find an attractor nor knowing γ. The key idea is to build on another measure, δ, that lower-bounds γ and is simpler to compute.

Definition 5.5. Let $T(\ell)$ be the total number of distinct substrings of length ℓ in T. Then

$$\delta = \max\{T(\ell)/\ell,\ \ell \geq 1\}.$$

Measure δ is related to the expression $d_\ell(w)/\ell$, used by Raskhodnikova et al. [2013] to approximate z. Analogously to their result [Raskhodnikova et al. 2013, Lem. 4], we have the following bound in terms of attractors.

Lemma 5.6. It always holds $\delta \leq \gamma$.

Proof. Since every length-ℓ substring of T must have a copy containing an attractor position, it follows that there are at most $\ell \cdot \gamma$ distinct such substrings, that is, $T(\ell)/\ell \leq \gamma$ for all ℓ. □

Lemma 5.7. Measure δ can be computed in $O(n)$ time and space from $T[1\,..\,n]$.

Proof. Computing δ boils down to computing $T(\ell)$ for all $1 \leq \ell \leq n$. This is easily computed from a suffix tree on T [Weiner 1973] (which is built in $O(n)$ time). We first initialize all the counters $T(\ell)$ at zero. Then we traverse the suffix tree: for each leaf with string depth ℓ we add 1 to $T(\ell)$, and for each non-root internal node with k children and string depth ℓ' we subtract $k-1$ from $T(\ell')$. Finally, for all the ℓ values, from $n-1$ to 1, we add $T(\ell+1)$ to $T(\ell)$. Thus, the leaves count the unique substrings they represent, and the latter step accumulates the leaves descending from each internal node. The value subtracted at internal nodes accounts for the fact that their k distinct children should count only once toward their parent. □

We now show that δ can be used as a replacement of γ to build the grammar.

Theorem 5.8. Let $T[1\,..\,n]$ have a minimum attractor of size γ. Then we can build a locally-balanced locally-consistent RLCFG of size $g = O(\gamma \log(n/\gamma))$ and height $O(\log n)$

that generates (only) T in $O(n)$ expected time and $O(n)$ working space, without knowing γ.

Proof. We carry out $\log n$ iterations instead of $\log(n/\gamma)$, and the grammar is still of size $O(\gamma \log(n/\gamma))$; the extra iterations add only $O(\gamma)$ to the size.

The only other place where we need to know γ is when applying Lemma 5.3, to check that the total length of the distinct blocks resulting from the parsing, using a randomly chosen permutation, is at most $2c \cdot \gamma$. A workaround to this problem is to use measure $\delta \leq \gamma$, which (unlike γ) can be computed efficiently.

To obtain a bound on the sum of the lengths of the blocks formed, we add up all the possible substrings multiplied by the probability that they become a block. Consider a substring $\hat{S}[1..\ell+3]$ of $\hat{T}$. Whether $\hat{S}$ occurs as a mapped block extension, that is, whether it occurs with $\hat{S}' = \hat{S}[3..\ell+2]$ being a block, depends only on π and $\hat{S}$, because by Lemma 3.11, if $\hat{S}'$ forms a block inside one occurrence of $\hat{S}$, it must form a block inside each occurrence of $\hat{S}$. Let us now consider the probability that $\hat{S}'$ forms a block. As in the proof of Lemma 5.3, $\hat{S}[3..\ell/2+2]$ must have an increasing sequence of π-values or $\hat{S}[\ell/2+3..\ell+2]$ must have a decreasing sequence of π-values, and this holds for at most two out of $(\ell/2)!$ permutations π.

Therefore, any distinct substring of length $\ell+3$ (of which there are $T(\ell+3) \leq (\ell+3)\delta$) contributes a block of length ℓ to the grammar size with probability at most $2/(\ell/2)!$ (note that we may be counting the same block several times within different block extensions). The total expected contribution to the grammar size is therefore $\sum_{\ell \geq 2}(\ell+3)\delta \cdot \ell \cdot 2/(\ell/2)! = O(\delta)$.

As in the proof of Lemma 5.3, given the expectation of the form $c \cdot \delta$, we can try out permutations until the total contribution to the grammar size is at most $2c \cdot \delta$. After $O(1)$ attempts, in expectation, we obtain a grammar of size $O(\delta) \subseteq O(\gamma)$ without knowing γ.

We repeat the same process for each text T_r, since we know from Theorem 5.4 that every T_r has an attractor of size at most 5γ, so the value δ_r we compute on T_r satisfies $\delta_r \leq 5\gamma$. The sizes of all texts T_r add up to $O(n)$. □

6. AN INDEX BASED ON OUR GRAMMAR

Let G be a locally-balanced RLCFG of r rules and size $g \geq r$ on text $T[1..n]$, formed with the procedure of Section 5, thus $g = O(\gamma \log(n/\gamma))$ with γ being the smallest size of an attractor of T. We show how to build an index of size $O(g)$ that locates the *occ* occurrences of a pattern $P[1..m]$ in time $O(m + (occ+1)\log^{\epsilon} n)$.

We make use of the parse tree and the partial parse-tree [Rytter 2003]. We call the latter "grammar tree" and extend the concept to RLCFGs.

Definition 6.1. For CFGs, the grammar tree is obtained by pruning the parse tree: all but the leftmost occurrence of each nonterminal is converted into a leaf and its subtree is pruned. Then the grammar tree has exactly one internal node per distinct nonterminal and the total number of nodes is $g+1$: r internal nodes and $g+1-r$ leaves. For RLCFGs, we treat rules $A \to A_1^s$ as $A \to A_1 A_1^{[s-1]}$, where the node labeled $A_1^{[s-1]}$ is always a leaf (A_1 may also be a leaf, if it is not the leftmost occurrence of A_1). Since we define the size of A_1^s as 2, the grammar tree is still of size $g+1$.

We will identify a nonterminal A with the only internal grammar tree node labeled A. When there is no confusion on the referred node, we will also identify terminal symbols a with grammar tree leaves.

We extend an existing approach to grammar indexing [Claude and Navarro 2012] to the case of our RLCFGs. We start by classifying the occurrences in T of a pattern $P[1..m]$ into primary and secondary.

Definition 6.2. The leaves of the grammar tree induce a partition of T into $f = g+1-r$ phrases. An occurrence of $P[1\,..\,m]$ at $T[t\,..\,t+m-1]$ is primary if the lowest grammar tree node deriving a range of T that contains $T[t\,..\,t+m-1]$ is internal (or, equivalently, the occurrence crosses the boundary between two phrases); otherwise it is secondary.

6.1. Finding the primary occurrences

Let nonterminal A be the lowest (internal) grammar tree node that covers a primary occurrence $T[t\,..\,t+m-1]$ of $P[1\,..\,m]$. Then, if $A \to A_1 \cdots A_s$, there exists some $i \in [1\,..\,s-1]$ and $q \in [1\,..\,m-1]$ such that (1) a suffix of $\exp(A_i)$ matches $P[1\,..\,q]$, and (2) a prefix of $\exp(A_{i+1}) \cdots \exp(A_s)$ matches $P[q+1\,..\,m]$. The idea is to index all the pairs $(\exp(A_i)^{rev}, \exp(A_{i+1}) \cdots \exp(A_s))$ and find those where the first and second component are prefixed by $(P[1\,..\,q])^{rev}$ and $P[q+1\,..\,m]$, respectively. Note that there is exactly one such pair per border between two consecutive phrases (or leaves in the grammar tree).

Definition 6.3. Let v be the lowest (internal) grammar tree node that covers a primary occurrence $T[t\,..\,t+m-1]$ of P, $[t\,..\,t+m-1] \subseteq proj(v)$. Let v_i be the leftmost child of v that overlaps $T[t\,..\,t+m-1]$, $[t\,..\,t+m-1] \cap proj(v_i) \neq \emptyset$. We say that node v is the parent of the primary occurrence $T[t\,..\,t+m-1]$ of P, and node v_i is its locus.

We build a multiset $\mathcal{G}$ of $f-1 = g-r$ string pairs containing, for every rule $A \to A_1 \cdots A_s$, the pairs $(\exp(A_i)^{rev}, \exp(A_{i+1}) \cdots \exp(A_s))$ for $1 \leq i < s$. The ith pair is associated with the ith child of the (unique) A-labeled internal node of the grammar tree. The multisets $\mathcal{X}$ and $\mathcal{Y}$ are then defined as projections of $\mathcal{G}$ to the first and second coordinate, respectively. We lexicographically sort these multisets, and represent each pair $(X, Y) \in \mathcal{G}$ by the pair (x, y) of the ranks of $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$, respectively. As a result, $\mathcal{G}$ can be interpreted as a subset of the two-dimensional integer grid $[1\,..\,g-r] \times [1\,..\,g-r]$.

Standard solutions [Claude and Navarro 2012] to find the primary occurrences in grammars consider the partitions $P[1\,..\,q] \cdot P[q+1\,..\,m]$ for $1 \leq q < m$. For each such partition, we search for $(P[1\,..\,q])^{rev}$ in $\mathcal{X}$ to find the range $[x_1\,..\,x_2]$ of symbols A_i whose suffix matches $P[1\,..\,q]$, search for $P[q+1\,..\,m]$ in $\mathcal{Y}$ to find the range $[y_1\,..\,y_2]$ of rule suffixes $A_{i+1} \cdots A_s$ whose prefix matches $P[q+1\,..\,m]$, and finally search the two-dimensional grid for all the points in the range $[x_1\,..\,x_2] \times [y_1\,..\,y_2]$. This retrieves all the primary occurrences whose leftmost intersected phrase ends with $P[1\,..\,q]$.

From the locus A_i associated with each point (x, y) found, and knowing q, we have sufficient information to report the position in T of this primary occurrence and all of its associated secondary occurrences; we describe this process in Section 6.4.

This arrangement follows previous strategies to index CFGs [Claude and Navarro 2012]. To include rules $A \to A_1^s$, we just index the pair $(\exp(A_1)^{rev}, \exp(A_1)^{s-1})$, which corresponds precisely to treating the rule as $A \to A_1 A_1^{[s-1]}$ to build the grammar tree. It is not necessary to index other positions of the rule, since their pairs will look like $(\exp(A_1)^{rev}, \exp(A_1)^{s'})$ with $s' < s-1$, and if $P[q+1\,..\,m]$ matches a prefix of $\exp(A_1)^{s'}$, it will also match a prefix of $\exp(A_1)^{s-1}$. The other occurrences inside $\exp(A_1)^{s-1}$ will be dealt with as secondary occurrences.

Finally note that, by definition, a pattern P of length $m = 1$ has no primary occurrences. We can, however, find all of its occurrences at the end of a phrase boundary by searching for $P[1\,..\,1]^{rev} = P[1]$ in $\mathcal{X}$, to find $[x_1\,..\,x_2]$, and assuming $[y_1\,..\,y_2] = [1\,..\,g-r]$. We can only miss the end of the last phrase boundary, but this is symbol \$, which (just as #) is not present in search patterns. We can just treat these points (x, y) as the primary occurrences of P, and report them and their associated secondary occurrences with the same mechanism we will describe for general patterns in Section 6.4.

A geometric data structure can represent our grid of size $(g-r) \times (g-r)$ with $g-r$ points in $O(g-r) \subseteq O(g)$ space, while performing each range search in time $O(\log^\epsilon g)$ plus $O(\log^\epsilon g)$ per primary occurrence found, for any constant $\epsilon > 0$ [Chan et al. 2011].

6.2. Parsing the pattern

In most previous work on grammar-based indexes, all the $m-1$ partitions $P = P[1\,..\,q] \cdot P[q+1\,..\,m]$ are tried out. We now show that, in our locally-consistent parsing, the number of positions that must be tried is reduced to $O(\log m)$.

Lemma 6.4. Using our grammar of Section 5, there are only $O(\log m)$ positions q yielding primary occurrences of $P[1\,..\,m]$. These positions belong to $M(P)+1$ (see Definition 4.8).

Proof. Let A be the parent of a primary occurrence $T[t\,..\,t+m-1]$, and let r be the round where A is formed. There are two possibilities:

(1) $A \to A_1 \cdots A_s$ is a block-forming rule, and for some $1 \le i < s$, a suffix of $\exp(A_i)$ matches $P[1\,..\,q]$, for some $1 \le q < m$. This means that $q-1 = \min \hat{B}_{r-1}(t, t+m-1)$.
(2) $A \to A_1^s$ is a run-length nonterminal, and a suffix of $\exp(A_1)$ matches $P[1\,..\,q]$, for some $1 \le q < m$. This means that $q-1 = \min B_r(t, t+m-1)$.

In either case, $q \in M(P)+1$ by Lemma 4.11. □

In order to construct $M(P)$ using Definitions 4.8 and 4.7, we need to already have an occurrence of P, which is not feasible in our context. Hence, we imagine parsing two texts, T and $P^* = \#P\$$, simultaneously using the permutations π_r we choose for T at each round r. It is easy to verify that the results of Sections 3 and 4 remain valid across substrings of both T and P^*, because they do not depend on how the permutations are chosen.

Hence, our goal is to parse P^* at query time in order to build $M(P)$ using the occurrence of P in P^*. We now show how to implement this step in $O(m)$ time. To carry out the parsing, we must preserve the permutations π_r of the alphabet used at each of the $O(\log n)$ rounds of the parsing of T, so as to parse P^* in the same way. The alphabets in each round are disjoint because all the blocks are of length 2 at least. Therefore the total size of these permutations coincides with the total number of terminals and nonterminals in the grammar, thus by Lemma 5.3 and Theorem 5.4 they require $O(\gamma)$ space per round and $O(g)$ space overall.

Let us describe the first round of the parsing. We first traverse $P^* = P_0^*$ left-to-right and identify the runs a^ℓ. Those are sought in a perfect hash table where we have stored all the first-round pairs (a, ℓ) existing in the text, and are replaced by their corresponding nonterminal $A \to a^\ell$ (see below for the case where a^ℓ does not appear in the text). The result of this pass is a new sequence $\hat{P}^* = \hat{P}_0^*$. We then traverse $\hat{P}^*$, finding the local minima (and thus identifying the blocks) in $O(m)$ time. For this, we have stored the values $\pi(a) = \pi_0(a)$ associated with each terminal a in another perfect hash table (for the nonterminals $A \to a^\ell$ just created, we have $\pi(A) = \pi(a)$; recall Section 3). To convert the identified blocks $A \to A_1 \cdots A_k$ into nonterminals for the next round, such tuples $(A_1 \cdots A_k)$ have been stored in yet another perfect hash table, from which the nonterminal A is obtained. This way, we can identify all the blocks in time $O(m)$, and proceed to the next round on the resulting sequence of nonterminals, P_1^*. The size of the first two hash tables is proportional to the number of terminals and nonterminals in the level, and the size of the tuples stores in the third table is proportional to the right-hand-sides of the rules created during the parsing. By Theorem 5.4, those sizes are $O(\gamma)$ per round and $O(g)$ added over all the rounds.

Since the grammar is locally balanced, P^* is parsed in $O(\log m)$ iterations, where at the rth iteration we parse P_{r-1}^* into a sequence of blocks whose total number is at most half of the preceding one, by Observation 3.4. Since we can find the partition into blocks in linear

time at any given level, the whole parsing takes time $O(m)$. Construction of the sets $B_r(P)$, $\hat{B}_r(P)$, and $M(P)$ from Definitions 4.7 and 4.8, respectively, also takes $O(m)$ time.

Note that P_r^* might contain blocks and runs that do not occur in T_r. By Lemma 4.10, if a block in P_r^* is not among the leftmost 4 or rightmost 2 blocks, then it must also appear within any occurrence of P in T, and as a result, the same must also be true for runs in P_{r+1}^*. Consequently, if a block (or a run) is not among those 6 extreme ones yet it does not appear in the hash table, we can abandon the search. As for the $O(1)$ allowed new blocks (and runs), we gather them in order to consistently assign new nonterminals and (in case of blocks) arbitrary unused π_r-values. We then proceed normally with subsequent levels of the parsing. Note that the newly formed blocks cannot appear anymore since distinct levels use distinct symbols, so we do not attempt to insert them into the perfect hash tables.

6.3. Searching for the pattern prefixes and suffixes

As a result of the previous section, we need only search for $\tau = O(\log m)$ (reversed) prefixes and suffixes of P in $\mathcal{X}$ or $\mathcal{Y}$, respectively. In this section we show that the corresponding ranges $[x_1 \,..\, x_2]$ and $[y_1 \,..\, y_2]$ can be found in time $O(m + \tau \log^2 m) = O(m)$. We build on the following result.

Lemma 6.5 (cf. [Belazzougui et al. 2010; Gagie et al. 2014; Gagie et al. 2018]). Let $\mathcal{S}$ be a set of strings and assume we have a data structure supporting extraction of any length-ℓ prefix of strings in $\mathcal{S}$ in time $f_e(\ell)$ and computation of a given Karp–Rabin signature κ of any length-ℓ prefix of strings in $\mathcal{S}$ in time $f_h(\ell)$. We can then build a data structure of $O(|\mathcal{S}|)$ words such that, later, we can solve the following problem in $O(m+\tau(f_h(m)+\log m)+f_e(m))$ time: given a pattern $P[1 \,..\, m]$ and $\tau > 0$ suffixes $Q_1, \ldots, Q_\tau$ of P, find the ranges of strings in (the lexicographically-sorted) $\mathcal{S}$ prefixed by $Q_1, \ldots, Q_\tau$.

Proof. The proof simplifies a lemma from Gagie et al. [2018, Lem 5.2].

First, we require a Karp–Rabin function κ that is collision-free between equal-length text substrings whose length is a power of two. We can find such a function at index construction time in $O(n \log n)$ expected time and $O(n)$ space [Bille et al. 2014]. We extend the collision-free property to pairs of equal-letter strings of arbitrary length by switching to the hash function κ' defined as $\kappa'(T[i \,..\, i+\ell-1]) = \langle \kappa(T[i \,..\, i+2^{\lfloor \log \ell \rfloor}-1]), \kappa(T[i+\ell-2^{\lfloor \log \ell \rfloor} \,..\, i+\ell-1]), \ell \rangle$.

Z-fast tries [Belazzougui et al. 2010, Sec. H.2] solve the weak part of the lemma in $O(m \log(\sigma)/w + \tau \log m)$ time. They have the same topology of a compact trie on $\mathcal{S}$, but use function κ' to find a candidate node for Q_i in time $O(\log |Q_i|) = O(\log m)$. We compute the κ'-signatures of all pattern suffixes $Q_1, \ldots, Q_\tau$ in $O(m)$ time, and then search the z-fast trie for the τ suffixes Q_i in time $O(\tau \log m)$.

By weak we mean that the returned answer for each suffix Q_i is not guaranteed to be correct if Q_i does not prefix any string in $\mathcal{S}$: we could therefore have false positives among the answers, though false negatives cannot occur. A procedure for discarding false positives [Gagie et al. 2014] requires extracting substrings and their signatures from $\mathcal{S}$. We describe and simplify this strategy in detail in order to analyze its time complexity in our scenario.

Let $Q_1, \ldots, Q_j$ be the pattern suffixes for which the z-fast trie found a candidate node. Order the pattern suffixes so that $|Q_1| < \cdots < |Q_j|$, that is, Q_i is a suffix of $Q_{i'}$ whenever $i < i'$. In addition, let $v_1, \ldots, v_j$ be the candidate nodes (explicit or implicit) of the z-fast trie such that all substrings below them are prefixed by $Q_1, \ldots, Q_j$ (modulo false positives), respectively, and let $t_i = \mathit{string}(v_i)$ be the substring read from the root of the trie to v_i. Our goal is to discard all nodes v_k such that $t_k \neq Q_k$.

Note that it is easy to check (in $O(\tau \cdot f_h(m))$ time) that $\kappa'(Q_i) = \kappa'(t_i)$ for all $i = 1, \ldots, j$. If a string t_i does not pass this test, then clearly v_i needs to be discarded because it must be the case that $Q_i \neq t_i$. We can thus safely assume that $\kappa'(Q_i) = \kappa'(t_i)$ for all $i = 1, \ldots, j$.

As a second simplification, we note that it is also easy to check (again in $O(\tau \cdot f_h(m))$ time) that t_a is a suffix of t_b whenever $1 \le a < b \le j$. Starting from $a = 1$ and $b = 2$, we check that $\kappa'(t_a) = \kappa'(t_b[|t_b| - |t_a| + 1\,..\,|t_b|])$. If the test succeeds, we know for sure that t_a is a suffix of t_b, since κ' is collision-free among text substrings: we increment $b \leftarrow b + 1$, set a to the next index such that v_a was not discarded (at the beginning of the procedure, no v_a has been discarded), and repeat. Otherwise, we clearly need to discard v_b since $\kappa'(Q_b[|t_b| - |t_a| + 1\,..\,|t_b|]) = \kappa'(Q_a) = \kappa'(t_a) \neq \kappa'(t_b[|t_b| - |t_a| + 1..|t_b|])$, therefore $Q_b \neq t_b$. Then, we discard v_b and increment $b \leftarrow b + 1$. From now on we can thus safely assume that t_a is a suffix of t_b whenever $1 \le a < b \le j$.

The last step is to compare explicitly t_j and Q_j in $O(f_e(m))$ time. Since we established that (i) t_a is a suffix of t_b whenever $1 \le a < b \le j$, (ii) by definition, Q_a is a suffix of Q_b whenever $1 \le a < b \le j$, and (iii) $|Q_i| = |t_i|$ for all $i = 1, \ldots, j$ (since function κ' includes the string's length and we know that $\kappa'(Q_i) = \kappa'(t_i)$ for all $i = 1, \ldots, j$), checking $t_j = Q_j$ is enough to establish that $t_i = Q_i$ for all $i = 1, \ldots, j$. However, $t_j \neq Q_j$ is not enough to discard all v_i: it could also be the case that only a proper suffix of t_j matches the corresponding suffix of Q_j, and some v_i pass the test. We therefore compute the longest common suffix s between t_j and Q_j, and discard only those v_i such that $|t_i| > s$.

To analyze the running time, note that we compute κ'-signatures of strings that are always suffixes of prefixes of length at most m of strings in $\mathcal{S}$ (because our candidate nodes $v_1, \ldots, v_j$ are always at depth at most m). By definition, to retrieve $\kappa'(t_i)$ we need to compute the two κ-signatures of the length-2^e prefix and suffix of t_i, for some $e \le \log |t_i| \le \log m$, $1 \le i \le j$. Computing the required κ'-signatures reduces therefore to the problem of computing κ-signatures of suffixes of prefixes of length at most m of strings in $\mathcal{S}$. Let $R' = t_b[|t_b| - s + 1\,..\,|t_b|]$ be such a length-s string of which we need to compute $\kappa(R')$. Then, $\kappa(R') = \kappa(t_b) - \kappa(t_b[1\,..\,|t_b| - s]) \cdot c^s \mod \mu$. Both signatures on the right-hand side are prefixes of suffixes of length at most m of strings in $\mathcal{S}$. The value $c^s \mod \mu$ can moreover be computed in $O(\log m)$ time using the fast exponentiation algorithm. It follows that, overall, computing the required κ'-signatures takes $O(f_h(m) + \log m)$ time per candidate node. For the last candidate, we extract the prefix t_j of length at most m ($O(f_e(m))$ time) of one of the strings in $\mathcal{S}$ and compare it with the longest candidate pattern suffix ($O(m)$ time). There are at most τ candidates, so the verification takes time $O(m + \tau \cdot (f_h(m) + \log m) + f_e(m))$. Added to the time to find the candidates in the z-fast trie, we obtain the claimed bounds. □

Therefore, when $\mathcal{S}$ is $\mathcal{X}$ or $\mathcal{Y}$, we need to extract length-ℓ prefixes of reverse phrases (i.e., of some $\exp(A_i)^{rev}$) or prefixes of consecutive phrases (i.e., of some $\exp(A_{i+1}) \cdots \exp(A_s)$) in time $f_e(\ell)$. The next result implies that we can obtain $f_e(\ell) = O(\ell)$.

Lemma 6.6 (cf. [Gasieniec et al. 2005], [Claude and Navarro 2012, Sec. 4.3]). Given a RLCFG of size g, there exists a data structure of size $O(g)$ such that any prefix or suffix of $\exp(A)$ can be obtained from any nonterminal A in real time.

Proof. Gasieniec et al. [2005] show how to extract any prefix of any $\exp(A)$ in a CFG of size g in Chomsky Normal Form, in real time, using a data structure of size $O(g)$. This was later extended to general CFGs [Claude and Navarro 2012, Sec. 4.3]. We now extend the result to RLCFGs.

Let us first consider prefixes. Define a forest of tries T_G with one node per distinct nonterminal or terminal symbol. Let us identify symbols with nodes of T_G. Terminal symbols are trie roots, and A_1 is the parent of A in T_G iff A_1 is the leftmost symbol in the rule that defines A, that is, $A \to A_1 \cdots$. For the rules $A \to A_1^s$, we also let A_1 be the parent of A. We augment T_G to support constant-time level ancestor queries [Bender and Farach-Colton 2004], which return the ancestor at a given depth of a given node. To extract ℓ symbols of $\exp(A)$, we start with the node A of T_G and immediately return the terminal a associated with its trie root (found with a level ancestor query). We now find the ancestor

of A at depth 2 (a child of the trie root). Let B be this node, with $B \to aB_2 \cdots B_s$. We recursively extract the symbols of $\exp(B_2)$ until $\exp(B_s)$, stopping after emitting ℓ symbols. If we obtain the whole $\exp(B)$ and still do not emit ℓ symbols, we go to the ancestor of A at depth 3. Let C be this node, with $C \to BC_2 \cdots C_r$, then we continue with $\exp(C_2)$, $\exp(C_3)$, and so on. At the top level of the recursion, we might finally arrive at extracting symbols from $\exp(A_2)$, $\exp(A_3)$, and so on. In this process, when we have to obtain the next symbols from a nonterminal $D \to E^s$, we treat it exactly as $D \to E \cdots E$ of size s, that is, we extract $\exp(E)$ $s-1$ further times.

Overall, we output ℓ symbols in time $O(\ell)$. The extraction is not yet real-time, however, because there may be several returns from the recursion between two symbols output. To ensure $O(1)$ time between two consecutive symbols obtained, we avoid the recursive call for the rightmost child of each nonterminal, and instead move to it directly.

Suffixes are analogous, and can be obtained in real-time in reverse order by defining a similar tree T'_G where A_s is the parent of A iff A_s is the rightmost symbol in the rule that defines A, $A \to \cdots A_s$. For rules $A \to A_1^s$, A_1 is still the parent of A. □

By slightly extending the same structures, we can compute any required signature in time $f_h(\ell) = O(\log^2 \ell)$ in our grammars.

Lemma 6.7. In the grammar of Section 5, we can compute Karp–Rabin signatures of prefixes of length ℓ of strings in $\mathcal{X}$ or $\mathcal{Y}$ in time $f_h(\ell) = O(\log^2 \ell)$.

Proof. Analogously as for extraction (Lemma 6.6), we consider the $O(\log \ell)$ levels of the grammar subtree containing the desired prefix. For each level, we find in $O(\log \ell)$ time the prefix/suffix of the rule contained in the desired prefix. Fingerprints of those prefixes/suffixes of rules are precomputed.

Strings in $\mathcal{X}$ are reversed expansions of nonterminals. Let every nonterminal X store the signatures of the reverses of all the suffixes of $\exp(X)$ that start at X's children. That is, if $X \to X_1 \cdots X_s$, store the signatures of $(\exp(X_i) \cdots \exp(X_s))^{rev}$ for all i. We use the trie T'_G of the proof of Lemma 6.6, where each trie node is a grammar nonterminal and its parent is the rightmost symbol of its defining rule. To extract the signature of the reversed prefix of length ℓ of a nonterminal X, we go to the node of X in T'_G and run an exponential search over its ancestors, so as to find in time $O(\log \ell)$ the lowest one whose expansion length is $\le \ell$. Let B be that nonterminal, then B is the first node in the rightmost path of the parse tree from X with $|B| \le \ell$. Note that the height of B is $O(\log \ell)$ because the grammar is locally balanced (Lemma 4.3), and moreover the parent $A \to B_1 \cdots B_{s-1}B$ of B satisfies $|A| > \ell$. We then exponentially search the preceding siblings of B until we find the largest i such that $|B_i| + \cdots + |B| > \ell$ (we must store these cumulative expansion lengths for each B_i). This takes $O(\log \ell)$ time. We collect the stored signature of $(\exp(B_{i+1}) \cdots \exp(B))^{rev}$; this is part of the signature we will assemble. Now we repeat the process from B_i, collecting the signature from the remaining part of the desired suffix. Since the depth of the involved nodes decreases at least by 1 at each step, the whole process takes $O(\log^2 \ell)$ time.

The case of $\mathcal{Y}$ is similar, now using the trie T_G of the proof of Lemma 6.6 and computing prefixes of signatures. The only difference is that we start from a given child Y_i of a nonterminal $Y \to Y_1 \cdots Y_t$ and the signature may span up to the end of Y. So we start with the exponential search for the leftmost Y_j such that $|Y_i| + \cdots + |Y_j| > \ell$; the rest of the process is similar.

When we have rules of the form $A \to A_1^s$, we find in constant time the desired copy A_i, from ℓ and $|A_1|$. Similarly, we can compute the signature κ of the last i copies of A_1 as $\kappa(\exp(A_1)^i) = \left(\kappa(\exp(A_1)) \cdot \frac{c^{|A_1|\cdot i}-1}{c^{|A_1|}-1}\right) \bmod \mu$: $c^{|A_1|} \bmod \mu$ and $(c^{|A_1|}-1)^{-1} \bmod \mu$ can be stored with A_1, and the exponentiation can be computed in $O(\log i) \subseteq O(\log \ell)$ time. □

Overall, we find the m ranges in the grid in time $O(m + \tau(f_h(m) + \log m) + f_e(m)) = O(m + \tau \log^2 m) = O(m + \log^3 m) = O(m)$, as claimed.

6.4. Reporting secondary occurrences

We report each secondary occurrence in constant amortized time, by adapting and extending an existing scheme for CFGs [Claude and Navarro 2012] to RLCFGs. Our data structure enhances the grammar tree with some fields per node v labeled A (where A is a terminal or a nonterminal):

(1) $v.anc = u$ is the nearest ancestor of v, labeled B, such that u is the root or B labels more than one node in the grammar tree. Note that, since u is internal in the grammar tree, it has the leftmost occurrence of label B in preorder. This field is undefined in the nodes labeled $A^{[s-1]}$ we create in the grammar tree (these do not appear in the parse tree).
(2) $v.\mathit{offs} = v_i - u_i$, where $\mathit{proj}(v) = [v_i \,..\, v_j]$ and $\mathit{proj}(u) = [u_i \,..\, u_j]$, is the offset of the projection $\exp(A)$ of v inside the projection $\exp(B)$ of u. This field is also undefined in the nodes labeled $A^{[s-1]}$.
(3) $v.\mathit{next} = v'$ is the next node in preorder labeled A, our *null* if v is the last node labeled A (those next appearances of A are leaves in the grammar tree). If $B \to A^s$, the internal node u labeled B has two children: v labeled A and v' labeled $A^{[s-1]}$. In this case, $v.\mathit{next} = v'$, and $v'.\mathit{next}$ points to the next occurrence of a node labeled A, in preorder.

Let u, labeled A, be the parent of primary occurrence of P, with $A \to A_1 \cdots A_s$, and v, labeled A_i, be its locus. The grid defined in Section 6.1 gives us the pointer to v. We then know that the relative offset of this primary occurrence inside A_i is $|A_i| - q + 1$. We then move to the nearest ancestor of v we have recorded, $u' = v.anc$, where the occurrence of P starts at offset $\mathit{offs} = |A_i| - q + 1 + v.\mathit{offs}$ (note that u' can be u or an ancestor of it). From now on, to find the offset of this occurrence in T, we repeatedly add $u'.\mathit{offs}$ to offs and move to $u' \leftarrow u'.anc$. When u' reaches the root, offs is the position in T of the primary occurrence.

At every step of this upward path to the root, we also take the rightward path to $u'' \leftarrow u'.\mathit{next}$. If $u'' \neq \mathit{null}$, we recursively report the copy of the primary occurrence inside u'', continuing from the same current value of offs we have for u'.

In other words, from the node $u' = v.anc$ we recursively continue by $u'.anc$ and $u'.\mathit{next}$, forming a binary tree of recursive calls. All the leaves of this binary tree that are "left" children (i.e., by $u'.anc$) reach the root of the grammar tree and report a distinct offset in T each time. The total number of nodes in this tree is proportional to the number of occurrences reported, and therefore the amortized cost per occurrence reported is $O(1)$.

In case $A \to A_1^s$, the internal grammar tree node u labeled A has two children: v labeled A_1 and $v' = v.\mathit{next}$ labeled $A_1^{[s-1]}$. If P has a primary occurrence where $P[1 \,..\, q]$ matches a suffix of $\exp(A_1)$, the grid will send us to the node v, where the occurrence starts at offset $|A_1| - q + 1$. This is just the leftmost occurrence of P within $\exp(A)$, with offset $|A_1| - q + 1$ as well. We must also report all the secondary occurrences inside $\exp(A)$, that is, all the offsets $i \cdot |A_1| - q + 1$, for $i = 1, 2, \ldots$ as long as $i \cdot |A_1| - q + m \leq s \cdot |A_1|$. For each such offset we continue the reporting from $u' = v.anc$, with offset $\mathit{offs} = i \cdot |A_1| - q + 1 + v.\mathit{offs}$.

We might also arrive at such a node v by a *next* pointer, in which case the occurrence of P is completely inside $\exp(A_1)$, with offset offs. In this case, we must similarly propagate all the other $s - 1$ copies of A_1 upwards, and then continue to the right. Precisely, we continue from $u' = v.anc$ and offset $\mathit{offs} + i \cdot |A_1| + v.\mathit{offs}$, for all $0 \leq i < s$. Finally, we continue rightward to node $v'.\mathit{next}$ and with the original value offs.

Our amortized analysis stays valid on these run-length nodes, because we still do $O(1)$ work per new occurrence reported (these are s-ary nodes in our tree of recursive calls).

6.5. Short patterns

All our data structures use $O(g)$ space. After parsing the pattern to find the $\tau = O(\log m)$ relevant cutting points q in time $O(m)$ (Section 6.2), and finding the τ grid ranges $[x_1 \,..\, x_2] \times [y_1 \,..\, y_2]$ by searching $\mathcal{X}$ and $\mathcal{Y}$ in time $O(m)$ as well (Section 6.3), we look for the primary and secondary occurrences. Finding the former requires $O(\log^\epsilon g)$ time for each of the τ ranges, plus $O(\log^\epsilon g)$ time per primary occurrence found (Section 6.1). The secondary occurrences require just $O(1)$ time each (Section 6.4). This yields total time $O(m + \log m \log^\epsilon g + occ \log^\epsilon g)$ to find the occ occurrences of $P[1 \,..\, m]$.

Next we show how to remove the additive term $O(\log m \log^\epsilon g)$ by dealing separately with short patterns: we use $O(\gamma)$ further space and leave only an additive $O(\log^\epsilon g)$-time term needed for short patterns that do not occur in T; we then further reduce this term.

The cost $O(\log m \log^\epsilon g)$ comes from the $O(\log m)$ geometric searches, each having a component $O(\log^\epsilon g)$ that cannot be charged to the primary occurrences found [Chan et al. 2011]. That cost, however, impacts on the total search complexity only for short patterns: it can be $\omega(m)$ only if $m = O(\ell)$, with $\ell = \log^\epsilon g \log\log g$.

We can then store sufficient information to avoid this cost for the short patterns. Since T has an attractor of size γ, there can be at most $\gamma\ell$ substrings of length ℓ crossing an attractor element, and all the others must have a copy crossing an attractor element. Thus, there are at most $\gamma\ell$ distinct substrings of length ℓ in T, and at most $\gamma\ell^2$ distinct substrings of length up to ℓ. We store all these substrings in a succinct perfect hash table H [Belazzougui et al. 2009], using the function κ' of Lemma 6.5 as the key. The associated value for each such substring are the $O(\log \ell) = O(\log\log g)$ split points q that are relevant for its search (Section 6.2) and have points in the corresponding grid range (Section 6.1). Since each partition position q can be represented in $O(\log \ell) = O(\log\log g)$ bits, we encode all this information in $O(\gamma\ell^2 \log^2 \ell)$ bits, which is $O(\gamma)$ space for any $\epsilon < \frac{1}{2}$. Succinct perfect hash tables require only linear-bit space on top of the stored data [Belazzougui et al. 2009], $O(\gamma\ell^2)$ bits in our case. Avoiding the partitions that do not produce any result effectively removes the $O(\log m \log^\epsilon g)$ additive term on the short patterns, because that cost can be charged to the first primary occurrence found.

Note, however, that function κ' is collision-free only among the substrings of T, and therefore there could be short patterns that do not occur in T but still are sent to a position in H that corresponds to a short substring of T (within $O(g)$ space we cannot afford to store a locus to disambiguate). To discard those patterns, we proceed as follows. If the first partition returned by H yields no grid points, then this was due to a collision with another pattern, and we can immediately return that P does not occur in T. If, on the other hand, the first partition does return occurrences, we immediately extract the text around the first one in order to verify that the substring is actually P. If it is not, then this is also due to a collision and we return that P does not occur in T.

Obtaining the locus v of the first primary occurrence from the first partition q takes time $O(\log^\epsilon g)$, and extracting m symbols around it takes time $O(m)$, by using Lemma 6.6 around v. Detecting that a short pattern P does not occur in T then costs $O(m + \log^\epsilon g)$.

We can slightly reduce this cost to $O(m + \log^\epsilon \gamma)$, as follows. Since $g = O(\gamma \log(n/\gamma))$, we have $\log^\epsilon g \in O(\log^\epsilon \gamma + \log\log(n/\gamma))$. Let $\ell' = \log\log(n/\gamma)$. We store all the $\gamma\ell'$ distinct text substrings of length ℓ' in a compact trie C, using perfect hashing to store the children of each node, and associating the locus v of a primary occurrence with each trie node. The internal trie nodes represent all the distinct substrings shorter than ℓ'. The compact trie C requires $O(\gamma\ell') \subseteq O(\gamma \log(n/\gamma))$ space. A search for a pattern of length $m \leq \ell'$ that does not occur in T can then be discarded in $O(m)$ time, by traversing C and then verifying the pattern around the locus. Thus the additive term $O(\log^\epsilon g)$ is reduced to $O(\log^\epsilon \gamma)$.

6.6. Construction

Theorem 5.4 shows that we can build a suitable grammar in $O(n)$ expected time and $O(g)$ working space, if we know γ. If not, Theorem 5.8 shows that the working space rises to $O(n)$.

The grammar tree is then easily built in $O(g)$ time by traversing the grammar top-down and left-to-right from the initial symbol, and marking nonterminals as we find them for the first time; the next times they are found correspond to leaves in the grammar tree, so they are not further explored. By recording the sizes $|A|$ of all the nonterminals A, we also obtain the positions where phrases start.

Let us now recapitulate the data structures used by our index:

(1) The grid of Section 6.1 where the points of $\mathcal{X}$ and $\mathcal{Y}$ are connected.
(2) The perfect hash tables storing the permutations π, the runs a^ℓ, and the blocks generated, for each round of parsing, used in Section 6.2.
(3) The z-fast tries on $\mathcal{X}$ and $\mathcal{Y}$, for Section 6.3. This includes finding a collision-free Karp–Rabin function κ'.
(4) The tries T_G and T'_G, provided with level-ancestor queries and with the Karp–Rabin signatures of all the prefixes and suffixes of $A_1 \cdots A_s$ for any rule $A \to A_1 \cdots A_s$.
(5) The extra fields on the grammar tree to find secondary occurrences in Section 6.4.
(6) The structures H and C for the short patterns, in Section 6.5

Navarro and Prezza [2019, Sec. 4] carefully analyze the construction cost of points 1 and 3:[6] The multisets $\mathcal{X}$ and $\mathcal{Y}$ can be built from a suffix array in $O(n)$ time and space, but also from a sparse suffix array in $O(n\sqrt{\log g})$ expected time and $O(g)$ space [Gawrychowski and Kociumaka 2017]; this time drops to $O(n)$ if we allow the output to be correct w.h.p. only. A variant of the grid structure of point 1 is built in $O(g\sqrt{\log g})$ time and $O(g)$ space [Belazzougui and Puglisi 2016]. The z-fast tries of point 3 are built in $O(g)$ expected time and space. However, ensuring that κ' is collision-free requires $O(n \log n)$ expected time and $O(n)$ space [Bille et al. 2014], which is dominant. Otherwise, we can build in $O(n)$ expected time and no extra space a signature that is collision-free w.h.p.

The structures of point 2 are of total size $O(g)$ and are already built in $O(g)$ expected time and space during the parsing of T. It is an easy exercise to build the structures of points 4 and 5 in $O(g)$ time; the level-ancestor data structure is built in $O(g)$ time as well [Bender and Farach-Colton 2004].

To build the succinct perfect hash table H of point 6, we traverse the text around the $g - r$ phrase borders; this is sufficient to spot all the primary occurrences of all the distinct patterns. There are at most $g\ell^2$ substrings of length up to ℓ crossing a phrase boundary, where $\ell = \log^\epsilon g \log\log g$. All their Karp–Rabin signatures κ' can be computed in time $O(g\ell^2)$ as well, and inserted into a regular hash table to obtain the $O(\gamma\ell^2)$ distinct substrings. We then build H on the signatures, in $O(\gamma\ell^2)$ expected time [Belazzougui et al. 2009]. Therefore, the total expected time to create H is $O(g\ell^2)$, whereas the space is $O(\gamma\ell^2)$ (we can obtain this space even without knowing γ, by progressively doubling the size of the hash table as needed).

This construction space can be reduced to $O(\gamma\ell)$ by building a separate table H_m for each distinct length $m \in [1\,..\,\ell]$. Further, since we can spend $O(m)$ time when searching for a pattern of length m, we can split H_m into up to m subtables $H_{m,i}$, which can then be built separately within $O(g)$ total space: We stop our traversal each time we collect g distinct substrings of length m, build a separate succinct hash table $H_{m,i}$ on those, and start afresh to build a new table $H_{m,i+1}$. Since there are at most $\gamma m \le gm$ distinct substrings, we will build at most m tables $H_{m,1}, \ldots, H_{m,m}$. Note that, in order to detect whether each

[6] Their w corresponds to our g: an upper bound to the number of phrases in T.

substring appeared previously, we must search all the preceding tables $H_{m,1}, \ldots, H_{m,i-1}$ for it, which raises the construction time to $O(g\ell^3)$. At search time, our pattern may appear in any of the m tables $H_{m,i}$, so we search them all in $O(m)$ time.

In order to compute the information on the partitions of each distinct substring, we can simulate its pattern search. Since we only need to find its relevant split points q (Section 6.2), their grid ranges (Section 6.3), and which of these are nonempty (Section 6.1), the total time spent per substring of length up to ℓ is $O(\ell + \log \ell \log^\epsilon \gamma) = O(\ell)$. Added over the up to $\gamma\ell^2$ distinct substrings, the time is $O(\gamma\ell^3)$. The whole process then takes $O(g\ell^3)$ expected time and $O(g)$ space. We enforce $\epsilon < \frac{1}{6}$ to keep the time within $O(g\sqrt{\log g})$.

We also build the compact trie C on all the distinct substrings of length $\ell' = \log\log(n/\gamma)$. We can collect their signatures κ' in $O(g\ell')$ time around phrase boundaries, storing them in a temporary hash table that collects at most $O(\gamma\ell')$ distinct signatures. For each such distinct signature we find, we insert the corresponding substring in C, recording its corresponding locus, in $O(\ell')$ time. The locus must also be recorded for the internal trie nodes v we traverse, if the substring represented by v also crosses the phrase boundary; this must happen for some descendant leaf of v because v must have a primary occurrence. Since we insert at most $\gamma\ell'$ distinct substrings, the total work on the trie is $O(\gamma\ell'^2)$. Then the expected construction time of C is $O(g\ell' + \gamma\ell'^2) \subseteq O(g\ell'^2) \subseteq O(\gamma \log(n/\gamma)(\log\log(n/\gamma))^2) \subseteq O(n)$. The construction space is $O(\gamma\ell') = O(\gamma \log\log(n/\gamma)) \subseteq O(\gamma \log(n/\gamma))$.

Note that we need to know γ to determine ℓ'. If we do not know γ, we can try out all the lengths, from $\ell' = \log\log(n/g)$ to $\log\log n$; note that the unknown correct value is in this range because $\gamma \leq g$. For each length, we build the structures to collect the distinct substrings of length ℓ, but stop if we exceed g distinct ones. Note that we cannot exceed g distinct substrings for $\ell' \leq \log\log(n/\gamma)$ because, in the grammar of Section 5, it holds that $g \geq \gamma \log(n/\gamma) \geq \gamma \log\log(n/\gamma) \geq \gamma\ell'$, and this is the maximum number of distinct substrings of length ℓ' we can produce. We therefore build the trie C for the value ℓ' such that the construction is stopped for the first time with $\ell' + 1$. This value must be $\ell' \geq \log\log(n/\gamma)$, sufficiently large to ensure the time bounds of Section 6.5, and sufficiently small so that the extra space is in $O(g)$. The only penalty is that we carry out ℓ' iterations in the construction of the hash table (the trie itself is built only after we find ℓ'), which costs $O(g\ell'^2)$ time. This is the same construction cost we had, but now ℓ' can be up to $\log\log n$; therefore the construction cost is $O(g(\log\log n)^2)$. The construction space stays in $O(g)$ by design.

The total construction cost is then $O(n \log n)$ expected time and $O(n)$ space, essentially dominated by the cost to ensure a collision-free Karp–Rabin signature.

Theorem 6.8. Let $T[1\,..\,n]$ have an attractor of size γ. Then, there exists a data structure of size $g = O(\gamma \log(n/\gamma))$ that can find the occ occurrences of any pattern $P[1\,..\,m]$ in T in time $O(m + \log^\epsilon \gamma + occ \log^\epsilon g) \subseteq O(m + (occ + 1)\log^\epsilon n)$ for any constant $\epsilon > 0$. The structure is built in $O(n \log n)$ expected time and $O(n)$ space, without the need to know γ.

An index that is correct w.h.p. can be built in $O(n + g\sqrt{\log g} + g(\log\log n)^2) \subseteq O(n + g\sqrt{\log g})$ expected time. If we know γ, such an index can be built with $O(\log(n/\gamma))$ expected left-to-right passes on T (to build the grammar) plus $O(\gamma \log(n/\gamma))$ main-memory space.

Finally, note that if we want to report only $k < occ$ occurrences of P, their locating time does not anymore amortize to $O(1)$ as in Section 6.4. Rather, extracting each occurrence requires us to climb up the grammar tree up to the root. In this case, the search time becomes $O(m + (k + 1)\log n)$.

6.7. Optimal search time

We now explore various space/time tradeoffs for our index, culminating with a variant that achieves, for the first, time, optimal search time within space bounded by an important

Table I. Space-time tradeoffs for searching within attractor-bounded space; formulas are slightly simplified (see the corresponding theorems and corollaries for the precise expressions).

Source	Space	Time
Baseline [Navarro and Prezza 2019]	$O(\gamma \log(n/\gamma))$	$O(m \log n + occ \log^\epsilon n)$
Theorem 6.8	$O(\gamma \log(n/\gamma))$	$O(m + (occ+1) \log^\epsilon n)$
Corollary 6.9	$O(\gamma \log n)$	$O(m + occ \log^\epsilon n)$
Corollary 6.10	$O(\gamma \log(n/\gamma) \log\log n)$	$O(m + (occ+1) \log\log n)$
Corollary 6.11	$O(\gamma \log n \log\log n)$	$O(m + occ \log\log n)$
Theorem 6.12	$O(\gamma \log(n/\gamma) \log^\epsilon n)$	$O(m + occ)$

family of repetitiveness measures. The tradeoffs are obtained by considering other data structures for the grid of Section 6.1 and for the perfect hash tables of Section 6.5. Table I summarizes the results in a slightly simplified form; the construction times stay as in Theorem 6.8.

A first tradeoff is obtained by discarding the table H of Section 6.5 and using only a compact trie C', now to store the locus of a primary occurrence and the relevant split points of each substring of length up to $\ell = \log^\epsilon g \log\log g$. This adds $O(\gamma\ell)$ to the space, but it allows verifying that the short patterns actually occurs in T in time $O(m)$ without using the grid. As a result, the additive term $O(\log^\epsilon \gamma)$ disappears from the search time.

As seen in Section 6.6, the extra construction time for C' is now $O(g\ell^2)$, plus $O(\gamma\ell^3)$ to compute the relevant split points. This is within the $O(g\ell^3)$ time bound obtained for Theorem 6.8. The construction space is $O(\gamma\ell)$, which we can assume to be $O(n)$ because it is included in the final index size; if this is larger than n then the result holds trivially by using instead a suffix tree on T.

Corollary 6.9. Let $T[1\,..\,n]$ have an attractor of size γ. Then, there exists a data structure of size $g = O(\gamma(\log(n/\gamma) + \log^\epsilon(\gamma \log(n/\gamma)) \log\log(\gamma \log(n/\gamma)))) \subseteq O(\gamma \log n)$ that can find the *occ* occurrences of any pattern $P[1\,..\,m]$ in T in time $O(m+occ \log^\epsilon g) \subseteq O(m+occ \log^\epsilon n)$ for any constant $\epsilon > 0$. The structure is built in $O(n \log n)$ expected time and $O(n)$ space, without the need to know γ.

By using $O(g \log\log g)$ space for the grid, the range queries run in time $O(\log\log g)$ per query and per returned item [Chan et al. 2011]. This reduces the query time to $O(m + \log m \log\log g + occ \log\log g)$, which can be further reduced with the same techniques of Section 6.5: The additive term can be relevant only if $m = O(\ell)$ with $\ell = \log\log g \log\log\log g$. We then store in H all the $\gamma\ell^2$ patterns of length up to ℓ, with their relevant partitions, using $O(\gamma\ell^2(\log\ell)^2) = O(\gamma(\log\log g)^2(\log\log\log g)^4)$ bits, which is $O(\gamma)$ space. We may still need $O(\log\log g)$ time to determine that a short pattern does not occur in T. By storing the patterns of length $\ell' = \log\log\log(n/\gamma)$ in trie C, this time becomes $O(\log\log\gamma)$.

The grid structure can be built in time $O(g \log g)$. The construction time for H and C is lower than in Section 6.6, because ℓ and ℓ' are smaller here.

Corollary 6.10. Let $T[1\,..\,n]$ have an attractor of size γ. Then, there exists a data structure of size $g = O(\gamma \log(n/\gamma) \log\log(\gamma \log(n/\gamma))) \subseteq O(\gamma \log(n/\gamma) \log\log n)$ that can find the *occ* occurrences of any pattern $P[1\,..\,m]$ in T in time $O(m + \log\log\gamma + occ \log\log g) \subseteq O(m+(occ+1) \log\log n)$. The structure is built in $O(n \log n)$ expected time and $O(n)$ space, without the need to know γ.

By discarding H and building C' on the substrings of length $\ell = \log\log g \log\log\log g$, we increase the space by $O(\gamma\ell^2)$ and remove the additive term in the search time. The construction time for the grid is still $O(g \log g)$, but that of C is within the bounds of Corollary 6.9, because ℓ is smaller here.

Corollary 6.11. Let $T[1..n]$ have an attractor of size γ. Then, there exists a data structure of size $g = O(\gamma(\log(n/\gamma)\log\log(\gamma\log(n/\gamma)) + (\log\log(\gamma\log(n/\gamma))\log\log\log(\gamma\log(n/\gamma)))^2)) \subseteq O(\gamma\log n\log\log n)$ that can find the *occ* occurrences of any pattern $P[1..m]$ in T in time $O(m+occ\log\log g) \subseteq O(m+occ\log\log n)$. The structure is built in $O(n\log n)$ expected time and $O(n)$ space, without the need to know γ.

Finally, a larger geometric structure [Alstrup et al. 2000] uses $O(g\log^\epsilon g)$ space, for any constant $\epsilon > 0$, and reports in $O(\log\log g)$ time per query and $O(1)$ per result. This yields $O(m+\log m\log\log g+occ)$ search time. To remove the second term, we again index all the patterns of length $m \le \ell$, for $\ell = \log\log g\log\log\log g$, of which there are at most $\gamma\ell^2$. Just storing the relevant split points q is not sufficient this time, however, because we cannot even afford the $O(\log\log g)$ time to query the nonempty areas.

Still, note that the search time can be written as $O(m+\ell+occ)$. Thus, we only care about the short patterns that, in addition, occur less than ℓ times, since otherwise the third term, $O(occ)$, absorbs the second. Storing all the occurrences of such patterns requires $O(\gamma\ell^2)$ space: An enriched version C'' of the compact trie C records the number of occurrences in T of each node. Only the leaves (i.e., the patterns of length exactly ℓ) store their occurrences (if they are at most ℓ). Since there are at most $\gamma\ell$ leaves, the total space to store those occurrences is $O(\gamma\ell^2)$, dominated by the grid size. Shorter patterns correspond to internal trie nodes, and for them we must traverse all the descendant leaves in order to collect their occurrences.

To handle a pattern P of length up to ℓ, then, we traverse C'' and verify P around its locus. If P occurs in T, we see if the trie node indicates it occurs more than ℓ times. If it does, we use the normal search procedure using the geometric data structure and propagating the secondary occurrences. Otherwise, its (up to ℓ) occurrences are obtained by traversing all the leaves descending from its trie node: if an internal node occurs less than ℓ times, its descendant leaves also occur less than ℓ times, so all the occurrences of the internal node are found in the descendant leaves. The search time is then always $O(m+occ)$.

The expected construction time of the geometric structure [Alstrup et al. 2000] is $O(g\log g)$, and its construction space is $O(g\log^\epsilon g)$. Note that if the construction space exceeds $O(n)$, then so does the size of our index. In this case, a suffix tree obtains linear construction time and space with the same search time. Thus, we can assume the construction space is $O(n)$.

The trie C'' is not built in the same way C is built in Section 6.6, because we need to record the number of occurrences of each string of length up to ℓ. We slide the window of length ℓ through the whole text T instead of only around phrase boundaries. We maintain the distinct signatures κ' found in a regular hash table, with the counter of how many times they appear in T. When a new signature appears, its string is inserted in C'', a pointer from the hash table to the corresponding trie leaf is set, and the list of occurrences of the substring is initialized in the trie leaf, with its first position just found. Further occurrence positions of the string are collected at its trie leaf, until they exceed ℓ, in which case they are deleted. Thus we spend $O(n)$ expected time in the hash table and collecting occurrences, plus $O(\gamma\ell^2)$ time inserting strings in C''. From the number of occurrences of each leaf we can finally propagate those counters upwards in the trie, in $O(\gamma\ell)$ additional time.

Theorem 6.12. Let $T[1..n]$ have an attractor of size γ. Then, there exists a data structure of size $O(\gamma\log(n/\gamma)\log^\epsilon(\gamma\log(n/\gamma))) \subseteq O(\gamma\log(n/\gamma)\log^\epsilon n)$, for any constant $\epsilon > 0$, that can find the *occ* occurrences of any pattern $P[1..m]$ in T in time $O(m+occ)$. The structure is built in $O(n\log n)$ expected time and $O(n)$ space, without the need to know γ.

7. COUNTING PATTERN OCCURRENCES

Navarro [2019] shows how an index like the one we describe in Section 6 can be used for counting the number of occurrences of $P[1\,..\,m]$ in T. First, he uses the result of Chazelle [1988] that a $p \times p$ grid can be enhanced by associating elements of any algebraic semigroup to the points, so that later we can aggregate all the elements inside a rectangular area in time $O(\log^{2+\epsilon} p)$, for any constant $\epsilon > 0$, with a structure using $O(p)$ space.[7] The structure is built in $O(p \log p)$ time and $O(p)$ space [Chazelle 1988]. Then, Navarro [2019] shows that one can associate with a CFG the number of secondary occurrences triggered by each point in a grid analogous to that of Section 6.1, so that their sums can be computed as described.

We now improve upon the space and time using our RLCFG of Section 6. Three observations are in order (cf. [Claude and Navarro 2012; Navarro 2019]):

(1) The occurrences reported are all those derived from each point (x, y) contained in the range $[x_1\,..\,x_2] \times [y_1\,..\,y_2]$ of each relevant partition $P[1\,..\,q] \cdot P[q+1\,..\,m]$.
(2) Even if the same point (x, y) appears in distinct overlapping ranges $[x_1\,..\,x_2] \times [y_1\,..\,y_2]$, each time it corresponds to a distinct value of q, and thus to distinct final offsets in T. Therefore, all the occurrences reported are distinct.
(3) The number of occurrences reported by our procedure in Section 6.4 depends only on the initial locus associated with the grid point (x, y). This will change with run-length nodes and require special handling, as seen later.

Therefore, we can associate with each point (x, y) in the grid (and with the corresponding primary occurrence) the total number of occurrences triggered with the procedure of Section 6.4. Then, counting the number of occurrences of a partition $P = P[1\,..\,q] \cdot P[q+1\,..\,m]$ corresponds to summing up the number of occurrences of the points that lie in the appropriate range of the grid.

As seen in Section 6.2, with our particular grammar there are only $O(\log m)$ partitions of P that must be tried in order to recover all of its occurrences. Therefore, we use our structures of Sections 6.1 to 6.3 to find the $O(\log m)$ relevant ranges $[x_1\,..\,x_2] \times [y_1\,..\,y_2]$, all in $O(m)$ time, and then we count the number of occurrences in each such range in time $O(\log^{2+\epsilon} p) \subseteq O(\log^{2+\epsilon} g)$. The total counting time is then $O(m + \log m \log^{2+\epsilon} g)$. When the second term dominates, $m \leq \log m \log^{2+\epsilon} g$, it holds $\log m \log^{2+\epsilon} g \in O(\log^{2+\epsilon} g \log\log g)$, which is $O(\log^{2+\epsilon} g)$ by infinitesimally adjusting ϵ.

Under the assumption that there are no run-length rules (we remove this assumption later), our counting time is then $O(m + \log^{2+\epsilon} g)$. This improves sharply upon the previous result [Navarro 2019] in space (because it builds the grammar on a Lempel–Ziv parse instead of on attractors) and in time (because it must consider all the $m - 1$ partitions of P).

To build the structure, we must count the number of secondary occurrences triggered from any locus v, and then associate it with every point (x, y) having v as its locus. More precisely, we will compute the number of times any node u occurs in the parse tree of T. The process corresponds to accumulating occurrences over the DAG defined by the pointers $u.anc$ and $u.next$ of the grammar tree nodes u. Initially, let the counter be $c(u) = 0$ for every grammar tree node u, except the root, where $c(root) = 1$. We now traverse all the nodes u in some order, calling $compute(u)$ on each. Procedure $compute(u)$ proceeds as follows: If $c(u) > 0$ then the counter is already computed, so it simply returns $c(u)$. Otherwise, it sets $c(u) = compute(u.anc) + compute(u.next)$, recursively computing the counters of the two nodes. Nodes $A \to A_1^s$ are special cases. If $u.next$ is of the form $A_1^{[s-1]}$, then the correct formula is $c(u) = s \cdot compute(u.anc) + compute(u.next.next)$. On the other hand, we do

[7] Navarro [2019] gives a simpler explicit construction for groups.

nothing for *compute*(u) if u is of the form $A_1^{[s-1]}$. The total cost is the number of edges in the DAG, which is 2 per grammar tree node, $O(g)$.

Finally, the counter of each point (x, y) associated with locus node v is the value $c(u)$, where u is the parent of v. A special case arises, however, if u corresponds to a run-length node $A \to A_1^s$, in which case the locus v is A_1. As seen in Section 6.4, the number of times u is reported is $s - \lceil (m-q)/|A_1| \rceil$, and therefore the correct counter to associate with (x, y) is $(s - \lceil (m-q)/|A_1| \rceil) \cdot c(u)$. The problem is that such a formula depends on $m-q$, so each point (x, y) could contribute differently for each alignment of the pattern. We then take a different approach for counting these occurrences.

Associated with loci A_1 with parent $A \to A_1^s$, instead of (x, y), we add to the grid the points $(x, y') = (\exp(A_1)^{rev}, \exp(A_1))$ with weight $c(u)$ and $(x, y'') = (\exp(A_1)^{rev}, \exp(A_1)^2)$ with weight $(s-2)c(u)$, extending the set $\mathcal{Y}$ so that it contains both $\exp(A_1)$ and $\exp(A_1)^2$. (Note that there could be various equal string pairs, which can be stored multiple times, or we can accumulate their counters.) We distinguish three cases.

(1) For the occurrences where $P[q+1\,..\,m]$ lies inside $\exp(A_1)$ (i.e., $m - q \le |A_1|$), the rule $A \to A_1^s$ is counted $c(u) + (s-2)c(u) = (s-1)c(u)$ times because both (x, y') and (x, y'') are in the range queried.
(2) For the occurrences where $P[q+1\,..\,m]$ exceeds the first $\exp(A_1)$ but does not span more than two (i.e., $|A_1| < m - q \le 2|A_1|$), the rule $A \to A_1^s$ is counted $(s-2)c(u)$ times because (x, y'') is in the range queried but (x, y') is not.
(3) For the occurrences where $P[q+1\,..\,m]$ spans more than two copies of $\exp(A_1)$, however, the rule $A \to A_1^s$ is not counted at all because neither (x, y') nor (x, y'') is in the range queried.

The key to handle the third case is that, if $P[1\,..\,q]$ spans a suffix of $\exp(A_1)$ and $P[q+1\,..\,m]$ spans at least two consecutive copies of $\exp(A_1)$, then it is easy to see that P is "periodic", $|A_1|$ being a "period" of P [Crochemore and Rytter 2003].

Definition 7.1. A string $P[1\,..\,m]$ has a period p if P consists of $\lfloor m/p \rfloor$ consecutive copies of $P[1\,..\,p]$ plus a (possibly empty) prefix of $P[1\,..\,p]$. Alternatively, $P[1\,..\,m-p] = P[p+1\,..\,m]$. The string P is periodic if it has a period $p \le m/2$.

We next show an important property relating periods and run-length nodes.

Lemma 7.2. Let there be a run-length rule $A \to A_1^s$ in our grammar. Then $|A_1|$ is the shortest period of $\exp(A)$.

Proof. Consider an A-labeled node v in the parse tree of T and let $proj(v) = [i\,..\,j]$ so that $T[i\,..\,j] = \exp(A)$. Denote the shortest period of $\exp(A)$ by p and note that $|A_1|$ is also a period of $\exp(A) = \exp(A_1)^s$. We conclude from the Periodicity Lemma [Fine and Wilf 1965] that $p = \gcd(p, |A_1|)$ and thus $d = |A_1|/p$ is an integer. For a proof by contradiction, suppose that $d > 1$. Let r denote the level of the run represented by v (so that A is a symbol in $\hat{T}_r$ and A_1 is a symbol in T_r).

Claim 7.3. For each level $r' \in [0\,..\,r]$, both $i+p-1$ and $j-p$ are level-r' block boundaries.

Proof. We proceed by induction on r'. The base case for $r' = 0$ holds trivially. Thus, consider a level $r' \in [1\,..\,r]$ and suppose that the claim holds for $r'-1$. By the inductive assumption, $T[i+p\,..\,j] = T[i\,..\,j-p]$ consist of full level-$(r'-1)$ blocks, so Lemma 4.5 yields $\hat{B}_{r'-1}(i+p, j) = \hat{B}_{r'-1}(i, j-p)$. Since $i + dp - 1$ is a level-r' block boundary, this set is non-empty and its minimum satisfies $\min \hat{B}_{r'-1}(i+p, j) < dp - p$. The final claim of Lemma 4.5 thus yields $B_{r'}(i+dp-p, j-p) = B_{r'}(i+dp, j)$. Consequently, since $i+dp-1$ is a level-r' block boundary, $p-1 \in B_{r'}(i+dp-p, j-p) = B_{r'}(i+dp, j)$, so $i+dp+p-1$ is also a level-r' block boundary. Iterating this reasoning $d(s-1)-2$ more times, we conclude

that $i+dp+2p-1, i+dp+3p-1, \ldots, j-p$ are all level-r' block boundaries. Moreover, Lemma 4.5 applied to $T[i\,..\,j-dp] = T[i+dp\,..\,j]$, which consist of full level-r' blocks, implies $p-1 \in B_{r'}(i, j-dp) = B_{r'}(i+dp, j)$, so $i+p-1$ is also a level-r' block boundary. □

Note that $T[i\,..\,j]$ consists of s full level-r blocks of length dp each. The claim instantiated to $r' = r$ contradicts this statement imposing blocks of length at most p at the extremities. □

Lemma 7.2 implies that, in the remaining case to be handled, the length $|A_1|$ must be precisely the shortest period of P.

Lemma 7.4. Let P be contained in $\exp(A)$ and contain two consecutive copies of $\exp(A_1)$, from rule $A \to A_1^s$. Then $|A_1|$ is the shortest period of P.

Proof. Clearly $|A_1|$ is a period of P because $P[1\,..\,m]$ is contained in a concatenation of strings $\exp(A_1)$; further, $|A_1| \le m/2$. Now assume P has a shorter period, $p < |A_1|$. Since $|A_1| + p < m$, P also has a period of length $p' = \gcd(|A_1|, p)$ [Fine and Wilf 1965]. This period is smaller than $|A_1|$ and divides it. Since P contains $\exp(A_1)$, this implies that $\exp(A_1)$, and thus $\exp(A)$, also have a period $p' < |A_1|$, contradicting Lemma 7.2. □

Therefore, all the run-length nonterminals $A \to A_1^s$, where A_1 is a locus of P with offset q and $m \ge 2|A_1|$, must satisfy $\exp(A_1) = P[q+1\,..\,q+p]$, where p is the shortest period of P. The shortest period p is easily computed in $O(m)$ time [Crochemore and Rytter 2003, Sections 1.7 and 3.1].

It is therefore sufficient to compute the Karp–Rabin fingerprints $k = \kappa'(\exp(A_1))$ (which we easily retrieve from the data we store for Lemma 6.7) for all the run-length rules $A \to A_1^s$, and store them in a perfect hash table with information on A_1. Let $s(A_1) = \{s \ge 3, A \to A_1^s\}$ be the different exponents associated with A_1. To each $s \in s(A_1)$, we associate two values

$$c(A_1, s) = \sum\{c(A) : A \to A_1^{s'}, s' \ge s\} \quad \text{and} \quad c'(A_1, s) = \sum\{s' \cdot c(A) : A \to A_1^{s'}, s' \ge s\}.$$

where $c(A)$ refers to $c(u)$ for the (only) internal grammar tree node u corresponding to nonterminal A. The total space to store the sets $s(A_1)$ and associated values is $O(g)$.

For each of the $O(\log m)$ relevant splits $P[1\,..\,q] \cdot P[q+1\,..\,m]$ obtained in Section 6.2, if $m-q > 2p$, then we look for $k = \kappa'(P[q+1\,..\,q+p])$ in the hash table. If we find it mapped to a non-terminal A_1, then we add $c'(A_1, s_{\min}) - c(A_1, s_{\min})\lceil (m-q)/p \rceil$ to the result, where $s_{\min} = \min\{s \in s(A_1), (s-1)|A_1| \ge m-q\}$. This ensures that each rule $A \to A_1^s$ with $s \ge 3$ and $|A_1|(s-1) \ge m-q$ is counted $(s - \lceil (m-q)/p \rceil) \cdot c(A)$ times. We find $s_{\min}$ by exponential search on $s(A_1)$ in $O(\log m)$ time, which over all the splits adds up to $O(\log^2 m)$.

Note that all the Karp–Rabin fingerprints for all the substrings of P can be computed in $O(m)$ time (see Section 6.3), and that we can easily rule out false positives: Lemma 6.5 filters out any decomposition of P for which $P[q+1\,..\,m]$ is not a prefix of any string $y \in \mathcal{Y}$. Since $\exp(A_1)^{s-1} \in \mathcal{Y}$ for every rule $A \to A_1^s$ and since $\mathcal{Y}$ consists of substrings of T, this guarantees that κ' does not admit any collision between $P[q+1\,..\,q+p]$ and a substring of T.

Theorem 7.5. Let $T[1\,..\,n]$ have an attractor of size γ. Then, there exists a data structure of size $g = O(\gamma \log(n/\gamma))$ that can count the number of occurrences of any pattern $P[1\,..\,m]$ in T in time $O(m + \log^{2+\epsilon} g) \subseteq O(m + \log^{2+\epsilon} n)$ for any constant $\epsilon > 0$. The structure can be built in $O(n \log n)$ expected time and $O(n)$ space, without the need to know γ.

An index that is correct w.h.p. can be built in $O(n + g \log g)$ expected time (the structures for secondary occurrences and for short patterns, Sections 6.4 and 6.5, are not needed). If we know γ, the index can be built in $O(\log(n/\gamma))$ expected left-to-right passes on T plus $O(g)$ main memory space.

Table II. Space-time tradeoffs for counting; formulas are slightly simplified (see the corresponding theorems for the precise expressions).

Source	Space	Time
Baseline [Navarro 2019]	$O(z \log(n/z))$	$O(m \log^{2+\epsilon} n)$
Theorem 7.5	$O(\gamma \log(n/\gamma))$	$O(m + \log^{2+\epsilon} n)$
Theorem 7.6	$O(\gamma \log(n/\gamma) \log n)$	$O(m)$

7.1. Optimal time

Chazelle [1988] offers other tradeoffs for operating the elements in a range, all very similar and with the same construction cost: $O(\log^2 p \log\log p)$ time and $O(p \log\log p)$ space, $O(\log^2 p)$ time and $O(p \log^\epsilon p)$ space. These yield, for our index, $O(m + (\log n \log\log n)^2)$ time and $O(g \log\log g)$ space, and $O(m + \log^2 n \log\log n)$ time and $O(g \log^\epsilon g)$ space.

If we use $O(p \log p)$ space, however, the cost to compute the sum over a range decreases significantly, to $O(\log p)$ [Willard 1985; Alstrup et al. 2000]. The expected construction cost becomes $O(p \log^2 p)$ [Alstrup et al. 2000]. Therefore, using $O(g \log g) \subseteq O(\gamma \log(n/\gamma) \log n)$ space, we can count in time $O(m+\log m \log g) \subseteq O(m+\log g \log\log g) \subseteq O(m + \log n \log\log n)$, which is yet another tradeoff.

More interesting is that we can reduce this latter time to the optimal $O(m)$. We index in a compact trie like C'' of Section 6.7 all the text substrings of length up to $\ell = 2 \log n \log(n/\gamma)$, directly storing their number of occurrences (but not their occurrence lists as in C''). Since there are $\gamma\ell$ distinct substrings of length ℓ, this requires $O(\gamma \log n \log(n/\gamma))$ space.

Consider our counting time $O(m+\log m \log n)$. If $\log(n/\gamma) \leq \log\log n$, then $\gamma \geq n/\log n$, and thus a suffix tree using space $O(n) = O(\gamma \log n)$ can count in optimal time $O(m)$. Thus, assume $\log(n/\gamma) > \log\log n$. The counting time can exceed $O(m)$ only if $m \leq \log m \log n$. In this case, since $m \leq \log m \log n \leq \log^2 n$, we have $m \leq 2 \log n \log\log n \leq 2 \log n \log(n/\gamma) = \ell$. All the queries for patterns of those lengths are directly answered using our variant of C'', in time $O(m)$, and thus our counting time is always $O(m)$.

We can still apply this idea if we do not know γ. Instead, we compute δ (recall Section 5.1) and use $\ell = 2 \log n \log(n/\delta)$. Since there are $T(\ell) \leq \delta\ell$ distinct substrings of length ℓ in T, the space for C'' is $O(\delta\ell) = O(\delta \log n \log(n/\delta)) \subseteq O(\gamma \log n \log(n/\gamma))$, the latter by Lemma 5.6. The reasoning of the previous paragraph then applies verbatim if we replace γ by δ.

The total space is then $O(g \log g + \gamma \log n \log(n/\gamma)) = O(\gamma \log n \log(n/\gamma))$. The construction cost of C'' is $O(n + \gamma \log^2 n \log^2(n/\gamma))$ time and $O(\gamma \log n \log(n/\gamma))$ space.[8] Alternatively we can obtain it by pruning the suffix tree of T in time and space $O(n)$. The cost to build the grid is $O(g \log^2 g) \subseteq (g \log^2 n)$. Note that, if $\gamma \log(n/\gamma) \log n > n$, we trivially obtain the result with a suffix tree; therefore the construction time of the grid is in $O(n \log n)$.

Theorem 7.6. Let $T[1..n]$ have an attractor of size γ. Then, there exists a data structure of size $O(\gamma \log(n/\gamma) \log n)$ that can count the number of occurrences of any pattern $P[1..m]$ in T in time $O(m)$. The structure can be built in $O(n \log n)$ expected time and $O(n)$ space, without the need to know γ.

If we know γ, then an index that is correct w.h.p. can be built in $O(g \log n)$ space apart from the passes on T, but we must build C'' without using a suffix tree, in additional time $O(\gamma \log^2 n \log^2(n/\gamma))$. Table II summarizes the results.

[8] If we use $\ell = 2 \log n \log(n/\delta)$, then C'' is built in $O(\delta \log^2 n \log^2(n/\delta)) \subseteq O(\gamma \log^2 n \log^2(n/\gamma))$ time and $O(\delta \log n \log(n/\delta)) \subseteq O(\gamma \log n \log(n/\gamma))$ space, because the costs increase with δ.

8. CONCLUSIONS

The size γ of the smallest string attractor of a text $T[1..n]$ is a recent measure of compressibility [Kempa and Prezza 2018] that is particularly well-suited to express the amount of information in repetitive text collections. It asymptotically lower-bounds many other popular dictionary-based compression measures like the size z of the Lempel–Ziv parse or the size g of the smallest context-free grammar generating (only) T, among many others. It is not known whether one can always represent T in compressed form in less than $\Theta(\gamma \log(n/\gamma))$ space, but within this space it is possible to offer direct access and reasonably efficient searches on T [Kempa and Prezza 2018; Navarro and Prezza 2019].

In this article we have shown that, within $O(\gamma \log(n/\gamma))$ space, one can offer much faster searches, in time competitive with, and in most cases better than, the best existing results built on other dictionary-based compression measures, all of which use $\Omega(z \log(n/z))$ space. By building on the measure γ, our results immediately apply to any index that builds on other dictionary measures like z and g. Our results are even competitive with self-indexes based on statistical compression, which are much more mature: we can locate the occ occurrences in T of a pattern $P[1..m]$ in $O(m + (occ + 1)\log^{\epsilon} n)$ time, and count them in $O(m + \log^{2+\epsilon} n)$ time, whereas the fastest statistically-compressed indexes obtain $O(m + occ \log^{\epsilon} n)$ time to locate and $O(m)$ time to count, in space proportional to the statistical entropy of T [Sadakane 2003; Belazzougui and Navarro 2014].

Further, we show that our results can be obtained without even knowing an attractor nor its minimum size γ. Rather, we can compute a lower bound $\delta \leq \gamma$ in linear time and use it to achieve $O(\gamma \log(n/\gamma))$ space without knowing γ. This is relevant because computing γ is NP-hard [Kempa and Prezza 2018]. Previous work [Navarro and Prezza 2019] assumed that, although they obtained indexes bounded in terms of γ, one would compute some upper bound on it, like z, to apply it in practice. With our result, we obtain data structures bounded in terms of γ without the need to find it.

Finally, we also obtain for the first time optimal search time using any index bounded by a dictionary-based compression measure. Within space $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$, for any constant $\epsilon > 0$, we can locate the occurrences in time $O(m + occ)$, and within $O(\gamma \log(n/\gamma) \log n)$ space we can count them in time $O(m)$. This is an important landmark, showing that it is possible to obtain the same optimal time reached by suffix trees in $O(n)$ space, now in space bounded in terms of a very competitive measure of repetitiveness. Such optimal time had also been obtained within space bounded by other measures that adapt to repetitiveness [Gagie et al. 2018; Belazzougui and Cunial 2017], but these are weaker than γ both in theory and in practice. Further, no statistical-compressed self-index using $o(n)$ space has obtained such optimal time.

As a byproduct, our developments yield a number of new or improved results on accessing and indexing on RLCFGs and CFGs; these are collected in Appendix A.

Future work. There are still several interesting challenges ahead:

— While one can compress any text T to $O(z)$ or $O(g)$ space (and even to smaller measures like $O(b)$ [Storer and Szymanski 1982]), it is not known whether one can compress it to $o(\gamma \log(n/\gamma))$ space. This is important to understand the nature of the concept of attractor and of measure γ.
— While one can support direct access and searches on T in space $O(g)$, it is not known whether one can support those in $o(z \log(n/z))$ or $o(\gamma \log(n/\gamma))$ space. Again, determining if this is a lower bound would yield a separation between γ, z, and g in terms of indexability.
— If we are given the size γ of some attractor, we can build our indexes in a streaming-like mode, with $O(\log(n/\gamma))$ expected passes on T plus main-memory space bounded in terms of γ, with high probability. This is relevant in practice when indexing huge text

collections. It would be important to do the same when no bound on γ is known. Right now, if we do not know γ, we need $O(n)$ extra space for a suffix tree that computes the measure $\delta \leq \gamma$.
— It is not clear if we can reach optimal search time in the "minimum" space $O(\gamma \log(n/\gamma))$, or what is the best time we can obtain in this case.
— The measure δ is interesting on its own, as it lower-bounds γ. It is interesting to find more precise bounds in terms of γ, and whether we can compress T, and even offer direct access and indexed searches on it, within space $O(\delta \log(n/\delta))$. There is some very recent work in this direction [Kociumaka et al. 2020].
— The fact that only $O(\log m)$ partitions of P are needed to spot all of its occurrences, which outperforms previous results [Nishimoto et al. 2019; Gawrychowski et al. 2018], was fundamental to obtain our bounds, and we applied them to counting in order to obtain optimal times as well. It is likely that this result is of even more general interest and can be used in other problems related to dictionary-compressed indexing and beyond.
— The result we obtain on counting pattern occurrences in $O(\gamma \log(n/\gamma))$ space is generalized to CFGs in Appendix A, but we could not generalize our result on specific RLCFGs to arbitrary ones. It is open whether this is possible or not.

Acknowledgements

Part of this work was carried out during the Dagstuhl Seminar 19241, "25 Years of the Burrows-Wheeler Transform". We also thank Travis Gagie for pointing us the early reference related to δ [Raskhodnikova et al. 2013] and Dmitry Kosolobov, who pointed out that a referenced result holds for constant alphabets only [Gagie et al. 2014]. Finally, we thank the reviewers for their thorough reading and useful remarks.

REFERENCES

Alstrup, S., Brodal, G. S., and Rauhe, T. 2000. New data structures for orthogonal range searching. In 41st Annual Symposium on Foundations of Computer Science, FOCS 2000. 198–207.

Belazzougui, D., Boldi, P., Pagh, R., and Vigna, S. 2010. Fast prefix search in little space, with applications. In 18th Annual European Symposium Algorithms, ESA 2010. LNCS Series, vol. 6346. 427–438.

Belazzougui, D., Botelho, F. C., and Dietzfelbinger, M. 2009. Hash, displace, and compress. In 17th Annual European Symposium Algorithms, ESA 2009. LNCS Series, vol. 5757. 682–693.

Belazzougui, D. and Cunial, F. 2017. Representing the suffix tree with the CDAWG. In 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017. LIPIcs Series, vol. 78. 7:1–7:13.

Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., and Raffinot, M. 2015. Composite repetition-aware data structures. In 26th Annual Symposium on Combinatorial Pattern Matching, CPM 2015. LNCS Series, vol. 9133. 26–39.

Belazzougui, D. and Navarro, G. 2014. Alphabet-independent compressed text indexing. ACM Transactions on Algorithms 10, 4, 23:1–23:19.

Belazzougui, D. and Puglisi, S. J. 2016. Range predecessor and Lempel–Ziv parsing. In 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016. 2053–2071.

Bender, M. A. and Farach-Colton, M. 2004. The level ancestor problem simplified. Theoretical Computer Science 321, 1, 5–12.

Bille, P., Ettienne, M. B., Gørtz, I. L., and Vildhøj, H. W. 2018. Time-space trade-offs for Lempel–Ziv compressed indexing. Theoretical Computer Science 713, 66–77.

Bille, P., Gørtz, I. L., Cording, P. H., Sach, B., Vildhøj, H. ., and Vind, S. 2017. Fingerprints in compressed strings. Journal of Computer and System Sciences 86, 171–180.

Bille, P., Gørtz, I. L., Sach, B., and Vildhøj, H. W. 2014. Time-space trade-offs for longest common extensions. Journal of Discrete Algorithms 25, 42–50.

Bille, P., Landau, G. M., Raman, R., Sadakane, K., Rao, S. S., and Weimann, O. 2015. Random access to grammar-compressed strings and trees. SIAM Journal on Computing 44, 3, 513–539.

Blumer, A., Blumer, J., Haussler, D., McConnell, R. M., and Ehrenfeucht, A. 1987. Complete inverted files for efficient text retrieval and analysis. Journal of the ACM 34, 3, 578–595.

Burrows, M. and Wheeler, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California.

Chan, T. M., Larsen, K. G., and Pătraşcu, M. 2011. Orthogonal range searching on the RAM, revisited. In 27th ACM Symposium on Computational Geometry, SoCG 2011. 1–10.

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

Chazelle, B. 1988. A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17, 3, 427–462.

Christiansen, A. R. and Ettienne, M. B. 2018. Compressed indexing with signature grammars. In 13th Latin American Symposium on Theoretical Informatics, LATIN 2018. LNCS Series, vol. 10807. 331–345.

Claude, F. and Navarro, G. 2012. Improved grammar-based compressed indexes. In 19th International Symposium on String Processing and Information Retrieval, SPIRE 2012. LNCS Series, vol. 7608. 180–192.

Crochemore, M. and Rytter, W. 2003. Jewels of Stringology. World Scientific.

Fine, N. J. and Wilf, H. S. 1965. Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society 16, 1, 109–114.

Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., and Puglisi, S. J. 2014. LZ77-based self-indexing with faster pattern matching. In 11th Latin American Symposium on Theoretical Informatics, LATIN 2014. LNCS Series, vol. 8392. 731–742.

Gagie, T., Navarro, G., and Prezza, N. 2018. Optimal-time text indexing in BWT-runs bounded space. In 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018. 1459–1477.

Gasieniec, L., Kolpakov, R. M., Potapov, I., and Sant, P. 2005. Real-time traversal in grammar-based compressed files. In 15th Data Compression Conference, DCC 2005. 458.

Gawrychowski, P. 2011. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In 19th Annual European Symposium on Algorithms, ESA 2011. LNCS Series, vol. 6942. 421–432.

Gawrychowski, P., Karczmarz, A., Kociumaka, T., Ącki, J., and Sankowski, P. 2018. Optimal dynamic strings. In 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018. 1509–1528.

Gawrychowski, P. and Kociumaka, T. 2017. Sparse suffix tree construction in optimal time and space. In 28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017. 425–439.

Kärkkäinen, J. and Ukkonen, E. 1996. Lempel-Ziv parsing and sublinear-size index structures for string matching. In 3rd South American Workshop on String Processing, WSP 1996. 141–155.

Karp, R. M. and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31, 2, 249–260.

Kempa, D. and Prezza, N. 2018. At the roots of dictionary compression: string attractors. In 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018. 827–840.

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

Kieffer, J. C. and Yang, E. 2000. Grammar-based codes: A new class of universal lossless source codes. IEEE Transactions on Information Theory 46, 3, 737–754.

Kociumaka, T., Navarro, G., and Prezza, N. 2020. Towards a definitive measure of repetitiveness. In Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN). To appear.

Kreft, S. and Navarro, G. 2013. On compressing and indexing repetitive sequences. Theoretical Computer Science 483, 115–133.

Lempel, A. and Ziv, J. 1976. On the complexity of finite sequences. IEEE Transactions on Information Theory 22, 1, 75–81.

McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. Journal of the ACM 23, 2, 262–272.

Mehlhorn, K., Sundar, R., and Uhrig, C. 1997. Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica 17, 2, 183–198.

Navarro, G. 2019. Document listing on repetitive collections with guaranteed performance. Theoretical Computer Science 772, 58–72.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. ACM Computing Surveys 39, 1.

Navarro, G. and Prezza, N. 2019. Universal compressed text indexing. Theoretical Computer Science 762, 41–50.

Nishimoto, T., I, T., Inenaga, S., Bannai, H., and Takeda, M. 2016. Fully dynamic data structure for LCE queries in compressed space. In 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016. LIPIcs Series, vol. 58. 72:1–72:15.

Nishimoto, T., I, T., Inenaga, S., Bannai, H., and Takeda, M. 2019. Dynamic index and LZ factorization in compressed space. Discrete Applied Mathematics.

Prezza, N. 2019. Optimal rank and select queries on dictionary-compressed text. In 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019. LIPIcs Series, vol. 128. 4:1–4:12.

Raskhodnikova, S., Ron, D., Rubinfeld, R., and Smith, A. D. 2013. Sublinear algorithms for approximating string compressibility. Algorithmica 65, 3, 685–709.

Rytter, W. 2003. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science 302, 1-3, 211–222.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 2, 294–313.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. 2015. Big data: Astronomical or genomical? PLOS Biology 13, 7, e1002195.

Storer, J. A. and Szymanski, T. G. 1982. Data compression via textual substitution. Journal of the ACM 29, 4, 928–951.

Weiner, P. 1973. Linear pattern matching algorithms. In 14th Annual Symposium on Switching and Automata Theory, SWAT 1973. 1–11.

Willard, D. E. 1985. New data structures for orthogonal range queries. SIAM Journal on Computing 14, 1, 232–253.

A. NEW RESULTS ON ARBITRARY RUN-LENGTH CONTEXT-FREE GRAMMARS

Along the article we have obtained a number of results for the specific RLCFG we build. Several of those can be generalized to arbitrary RLCFGs, leading to the same state of the art that CFGs now enjoy. We believe it is interesting to explicitly state those new results in general form: not only RLCFGs are always smaller than CFGs (and the difference can be asymptotically relevant, as in text $T = a^n$), but also our results in this article require space $O(\gamma \log(n/\gamma))$, whereas there always exists a RLCFG of size $g_{rl} = O(\gamma \log(n/\gamma))$. Indexes of size $O(g_{rl})$ have then the potential to be smaller than those built on attractors (e.g., $T = a^n$ is generated by a RLCFG of size $O(1)$, whereas $\gamma \log(n/\gamma) = \Theta(\log n)$).

A.1. Extracting substrings

The following result exists on CFGs [Bille et al. 2015]. They present their result on straight-line programs (SLPs, i.e., CFGs where right-hand sides are two nonterminals or one terminal symbol). While any CFG of size g can be converted into an SLP of size $O(g)$, we start by describing their structure generalized to arbitrary CFGs, which may be interesting when the grammar cannot be modified for some reason. We then show how to handle run-length rules $A \to A_1^s$ in order to generalize the result to RGCFGs.

Theorem A.1. Let a RLCFG of size g_{rl} generate (only) $T[1\,..\,n]$. Then there exists a data structure of size $O(g_{rl})$ that extracts any substring $T[p\,..\,p+\ell-1]$ in time $O(\ell + \log n)$.

Consider the parse tree $\mathcal{T}$ of $T[1\,..\,n]$. A heavy path starting at a node $v \in \mathcal{T}$ with children $v_1, \ldots, v_s$ chooses the child v_i that maximizes $|v_i|$, and continues by v_i in the same way, up to reaching a leaf. We say that v_i is the heavy child of v and define $h(v) = v_i$. The edge connecting v with its heavy child v_i is said to be heavy; those connecting v with its other children are light. Note that, if $v_j \neq h(v)$, then $|v_j| \leq |v|/2$; otherwise v_j would be the heavy child of v. Then, every time we descend by a light edge, the lenght of the node halves, and as a consequence no path from the root to a leaf may include more than $\log n$ light edges. A decomposition into heavy paths consists of the heavy path starting at the root of $\mathcal{T}$ and, recursively, all those starting at the children by light edges.

A.1.1. Accessing $T[p]$. For every internal node v with children $v_1, \ldots, v_s$ we define the starting positions of its children as $p_1(v) = 1$, $p_i(v) = p_{i-1}(v) + |v_{i-1}|$, for $2 \leq i \leq s$, and $p_{s+1} = |v| + 1$. We then store the set $C(v) = \{p_1(v), p_2(v), \ldots, p_{s+1}(v)\}$. Let us define $c(v) = p_i(v)$, where $v_i = h(v)$, as the starting position of the heavy child of v. Then, if v roots a heavy path $v = v^0, v^1, \ldots, v^k$, where $v^j = h(v^{j-1})$ for $1 \leq j \leq k$, and v^k is a leaf, we define

the starting positions in the heavy path as $s_1(v) = c(v)$ and $s_j(v) = s_{j-1}(v) - 1 + c(v^{j-1})$ for $2 \leq j \leq k$, and the ending positions as $e_j(v) = s_j(v) + |v^j|$ for $1 \leq j \leq k$. We then associate with v the increasing set $P(v) = \{s_1(v), s_2(v), \ldots, s_k(v), e_k(v), \ldots, e_2(v), e_1(v)\}$; note $e_k(v) = s_k(v) + 1$.

To find $T[p]$, we start at the root v of $\mathcal{T}$ (so $1 \leq p \leq |v|$) with children $v_1, \ldots, v_s$. We make a predecessor search on $C(v)$ to determine that $p_i(v) \leq p < p_{i+1}(v)$. If $v_i \neq h(v)$, we traverse the light edge to v_i and continue the search from v_i with $p \leftarrow p - p_i(v) + 1$. Otherwise, since $v_i = h(v)$, it holds that $p \geq p_i(v) = c(v) = s_1(v)$ and $p < p_{i+1}(v) = c(v) + |h(v)| = e_1(v)$. We then jump to the proper node in the heavy path that starts in v by making a predecessor search for p in $P(v)$. If we determine that $s_j(v) \leq p < s_{j+1}(v)$ or that $e_{j+1}(v) \leq p < e_j(v)$, we continue the search from v^j and $p \leftarrow p - s_j(v) + 1$. Otherwise, $p = s_k(v)$ and the answer is the terminal symbol associated with the leaf v^k. Note that, when we continue from v^j, this is not the head of a heavy path, but after searching $C(v^j)$ we are guaranteed to continue by a light edge. In each step, then, we perform two predecessor searches and traverse a light edge.

Bille et al. [2015] describe a predecessor data structure that, when finding the predecessor of x in a universe of size u, takes time $O(\log(u/(x^+ - x^-)))$, where x^+ and x^- are the predecessor and successor of x, respectively. Thus, when finding v_i in $C(v)$, this structure takes time $O(\log(|v|/|v_i|))$. If v_i is a light child, we continue by v_i, so the sum over all the light edges traversed telescopes to $O(\log |v|)$. When we descend to the heavy child, instead, we also find the node v^j in $P(v)$, which costs $O(\log(|v|/(s_{j+1}(v) - s_j(v) + 1))) = O(\log(|v|/c(v^j)))$ if $s_j(v) \leq p < s_{j+1}(v)$, or $O(\log(|v|/(e_j(v) - e_{j+1}(v) + 1))) = O(\log(|v|/(|v^j| - (c(v^j) + |h(v^j)|))))$ if $e_{j+1}(v) \leq p < e_j(v)$, or $O(\log |v|)$ if $p = s_k(v)$ (but this happens only once along the search). In the first two cases, we descend to v^j, which always starts descending by a light edge to some v_i^j at cost $O(\log(|v^j|/|v_i^j|))$. Since $|v_i^j| \leq c(v^j)$ (if $s_j(v) \leq p < s_{j+1}(v)$) or $|v_i^j| \leq |v^j| - (c(v^j) + |h(v^j)|)$ (if $e_{j+1}(v) \leq p < e_j(v)$), we can upper bound the cost to search $P(v)$ by $O(\log(|v|/|v_i^j|))$, and the cost to search $C(v^j)$ by $O(\log(|v^j|/|v_i^j|)) \subseteq O(\log(|v|/|v_i^j|))$ too, and then we continue the search from v_i^j. Therefore the cost also telescopes to $O(\log |v|)$ when we search a heavy path. Overall, the cost from the root of the parse tree is $O(\log n)$.

The remaining problem is that the structure is of size $O(|\mathcal{T}|) = O(n)$, but it can be made $O(g)$ as follows. The subtrees of $\mathcal{T}$ rooted by all the nodes v labeled with the same nonterminal A are identical, so in all of them the node $h(v)$ has the same label, say the terminal or nonterminal A_i. Bille et al. [2015] define a forest $\mathcal{F}$ with exactly one node $v(X) \in \mathcal{F}$ for each nonterminal or nonterminal X. If $v \in \mathcal{T}$ is labeled A and $h(v) \in \mathcal{T}$ is labeled A_i, then $v(A_i)$ is the parent of $v(A)$ in $\mathcal{F}$. The nodes $v(a)$ for terminals a are roots in $\mathcal{F}$. A heavy path from $v \in \mathcal{T}$, with v labeled A, then corresponds to an upward path from $v(A) \in \mathcal{F}$.

The sets $C(v)$ also depend only on the label A of $v \in \mathcal{T}$, so we associate them to the corresponding nonterminal A. The sizes of all sets $C(A)$ add up to the grammar size, because $C(A)$ has $s+1$ elements if the rule that defines A is of the form $A \to A_1 \cdots A_s$.[9] The sets $P(v)$ also depend only on the label A of $v \in \mathcal{T}$, but they are not stored completely in A. Instead, each node $v(A) \in \mathcal{F}$, corresponding to the nodes $v \in \mathcal{T}$ labeled A, and with parent $v(A_i) \in \mathcal{F}$, stores values $s(v(A)) = s(v(A_i)) + c(v) - 1$ and $e(v(A)) = e(v(A_i)) + |v| - c(v) - |h(v)| + 1$. For the roots $v(a) \in F$, we set $s(v(a)) = e(v(a)) = 0$. They then build two data structures for predecessor queries on tree paths, one on the $s(\cdot)$ and one on the $e(\cdot)$ values, which obtain the same complexities as on arrays. In order to find a position p from $v(A)$, we also store the position $p(A)$ in
$exp(A)$ of the root in $\mathcal{F}$ from where $v(A)$ descends, as well as the character

[9] To have the grammar size count only right-hand sides, rules $A \to \varepsilon$ must be removed or counted as size 1.

$exp(A)[p(A)]$. If $p = p(A)$, we just return that symbol and finish. Otherwise, if $p < p(A)$, we search for $p(A) - p$ in the fields $s(\cdot)$ from $v(A)$ to the root, finding $s(v(B)) \geq p(A) - p > s(v(B_i))$, with $v(B_i)$ the parent of $v(B)$ in $\mathcal{F}$. Otherwise, $p > p(A)$ and we search for $p - p(A)$ in the fields $e(\cdot)$ from $v(A)$ to the root, finding $e(v(B)) \geq p - p(A) > e(v(B_i))$, with $v(B_i)$ the parent of $v(B)$ in $\mathcal{F}$. In both cases, we must exit the heavy path from the node $v(B)$, adjusting $p \leftarrow p - s(v(A)) + s(v(B))$.

A.1.2. Extracting $T[p\,..\,q]$. To extract $T[p\,..\,q]$ in time $O(q - p + \log n)$, we store additional information as follows. In each heavy path $v^0, \ldots, v^k$, each node v^j stores a pointer $r(v^j) = h(v^t)$, where $j < t \leq k$ is the smallest value for which $h(v^t)$ is not the rightmost child of v^t. Similarly, $l(v^j) = h(v^t)$ for the smallest $j < t \leq k$ for which $h(v^t) > 1$. At query time, we apply the procedures to retrieve $T[p]$ and $T[q]$ simultaneously until they split at a node v^*, where $T[p]$ descends from the child v^*_i and $T[q]$ from the child v^*_j. Then the symbols $T[p\,..\,q]$ are obtained by traversing, in left-to-right order, (1) the children $v_{i+1}, \ldots$ of every light edge leading to v_i in the way to $T[p]$; (2) every sibling to the right of $r(v)$ for the nodes $v \in \{v_1, r(v_1), r(r(v_1)), \ldots\}$ for every v_1 rooting a heavy path in the way to $T[p]$; (3) the children $\{v^*_{i+1}, \ldots, v^*_{j-1}\}$ of v^*; (4) the children $v_1, \ldots, v_{i-1}$ of every light edge v_i in the way to $T[q]$; (5) every sibling to the left of $l(v)$ for the nodes $v \in \{v_1, l(v_1), l(l(v_1)), \ldots\}$ for every v_1 rooting a heavy path in the way to $T[q]$. For all those nodes, we traverse their subtrees completely to obtain chunks of $T[p\,..\,q]$ in optimal time (unless there are unary paths in the grammar, which can be removed or skipped with the information on $r(\cdot)$ or $l(\cdot)$). The left-to-right order between nodes in (1) and (2), and in (3) and (4), is obtained as we descend to $T[p]$ or $T[q]$. Finally, v^* is easily determined if it is the target of a light edge. Otherwise, if we exit a heavy path by distinct nodes v_p and v_q, then v^* is the highest of the two.

A.1.3. Extending to RLCFGs. The idea to include rules $A \to A_1^s$ is to handle them exactly as if they were $A \to A_1 \cdots A_1$, but using $O(1)$ space instead of $O(s)$. When v is labeled A and this is defined as $A \to A_1^s$, we would have a tie in determining the heavy child $h(v)$. We then act as if we chose the first copy of A_1, $h(v) = v_1$; in particular $v(A_1)$ is the parent of $v(A)$ in $\mathcal{F}$. If we have to descend by another child of v to reach position p inside v, we choose v_i with $i = \lceil p/|v_1| \rceil$ and set $p \leftarrow p - (i-1) \cdot |v_1|$, so we do not need to store the set $C(A)$ (which would exceed our space budget).

No pointer $l(v^j)$ will point to $h(v)$, but pointers $r(v^j)$ will. The pointers $r(v^j) = h(v^t)$ are actually stored as a pair (v^t, i) where $v^s_i = h(v^t)$; this allows accessing preceding and following siblings easily. With this format, we can also refer to the ith child of a run-length node and handle it appropriately.

A.2. Extracting prefixes and suffixes

The following result also exists on CFGs [Gasieniec et al. 2005], who use leftmost or rightmost paths instead of heavy paths. In our Lemma 6.6 we have extended it to arbitrary RLCFGs as well, without setting any restriction on the grammar.

Theorem A.2. Let a RLCFG of size g_{rl} generate (only) $T[1\,..\,n]$. Then there exists a data structure of size $O(g_{rl})$ that extracts any prefix or suffix of the expansion $\exp(A)$ of any nonterminal A in real time.

A.3. Computing fingerprints

The following result, already existing on CFGs [Bille et al. 2017], can also be extended to arbitrary RLCFGs. Note that it improves our Lemma 6.7 to $O(\log \ell)$ time, though we opted for a simpler variant in the body of the article.

Theorem A.3. Let a RLCFG of size g_{rl} generate (only) $T[1\,..\,n]$. Then there exists a data structure of size $O(g_{rl})$ that computes the Karp-Rabin signature of any substring $T[p\,..\,q]$ in time $O(\log n)$.

Recall that, given the signatures $\kappa(S_1)$ and $\kappa(S_2)$, one can compute the signature of the concatenation, $\kappa(S_1 \cdot S_2) = (\kappa(S_1) + c^{|S_1|} \cdot \kappa(S_2)) \bmod \mu$. One can also compute the signature of S_2 given those of S_1 and $S_1 \cdot S_2$, $\kappa(S_2) = ((\kappa(S_1 \cdot S_2) - \kappa(S_1)) \cdot c^{-|S_1|}) \bmod \mu$, and the signature of S_1 given those of S_2 and $S_1 \cdot S_2$, $\kappa(S_1) = (\kappa(S_1 \cdot S_2) - \kappa(S_2) \cdot c^{|S_1|}) \bmod \mu$. To have the terms $c^{\pm|S_1|}$ handy, we redefine signatures $\kappa(S)$ as triples $(\kappa(S), c^{|S|} \bmod \mu, c^{-|S|} \bmod \mu)$, which are easily maintained across the described operations.

We now show how to compute a fingerprint $\kappa(T[p\,..\,q])$ in $O(\log n)$ time on an arbitrary RLCFG. We present the current result [Bille et al. 2017], extended to general CFGs, and then include run-length rules.

We follow the idea of our Lemma 6.7, but combine it with heavy paths. Since we can obtain $\kappa(T[p\,..\,q])$ from $\kappa(T[1\,..\,q])$ and $\kappa(T[1\,..\,p-1])$, we only consider computing fingerprints of text prefixes. We associate with each nonterminal $A \to A_1 \cdots A_s$ the s signatures $K_i(A) = \kappa(\exp(A_1) \cdots \exp(A_{i-1}))$, for $1 \le i \le s$. We also associate signatures to nodes $v(A)$ in $\mathcal{F}$, $K(v(A)) = \kappa(\exp(A)[1\,..\,p(A)-1])$. Those values fit in $O(g)$ space.

To compute $\kappa(T[1\,..\,p])$ we start with $\kappa = 0$ and follow the same process as for accessing $T[p]$ in Section A.1. In our way, every time we descend by a light edge from v to v_i, where v is labeled A, we update $\kappa \leftarrow (\kappa + K_i(A) \cdot c^{|A_1|+\cdots+|A_{i-1}|}) \bmod \mu$. Note that the power of c is implicitly stored together with the signature $K_i(A)$ itself.

Instead, when we descend from $v(A)$ to $v(B)$ because $s(v(B)) \ge p(A) - p > s(v(B_i))$ or $e(v(B)) \ge p - p(A) > e(v(B_i))$, we first compute the signature κ' of the prefix of $\exp(A)$ that precedes $\exp(B)$, which is of length $\ell = s(v(A)) - s(v(B))$, and then update $\kappa \leftarrow \kappa \cdot c^{\ell} + \kappa'$ so as to concatenate that prefix (again, c^{ℓ} is computed together with κ'). We compute κ' from $K(v(B)) = \kappa(\exp(B)[1\,..\,p(B)-1])$ and $K(v(A)) = \kappa(\exp(A)[1\,..\,p(A)-1])$. Because $\exp(A)[p(A)]$ is the same symbol of $\exp(B)[p(B)]$, $\exp(B)[1\,..\,p(B)-1]$ is a suffix of $\exp(A)[1\,..\,p(A)-1]$. We then use the method to extract $\kappa(S_1)$ from $\kappa(S_1 \cdot S_2)$ and $\kappa(S_2)$.

When we arrive at $T[p]$, we include that symbol and have computed $\kappa = \kappa(T[1\,..\,p])$. The time is the same $O(\log n)$ required to access $T[p]$.

A.3.1. Handling run-length rules. The proof of Lemma 6.7 already shows how to handle run-length rules $A \to A_1^s$: we again treat them as $A \to A_1 \cdots A_1$. The only complication is that now we cannot afford to store the values $K_i(A)$ used to descend by light edges, but we can compute them as $K_i(A) = \kappa(\exp(A_1)^{i-1}) = \left(\kappa(\exp(A_1)) \cdot \frac{c^{|A_1| \cdot (i-1)} - 1}{c^{|A_1|} - 1}\right) \bmod \mu$: $c^{|A_1|} \bmod \mu$ and $(c^{|A_1|} - 1)^{-1} \bmod \mu$ can be stored with A_1, and the exponentiation can be computed in time $O(\log i) \subseteq O(\log s)$. Note that this is precisely the $O(\log(|v|/|v_i|))$ time we are allowed to spend when moving from node v to its child v_i by a light edge.

A.4. Locating pattern occurrences

Claude and Navarro [2012, Cor. 1] obtain a version of the following result that holds only for CFGs and offers search time $O(m^2 + (m + occ)\log^{\epsilon} n)$. We improve their complexity and generalize it to RLCFGs.

Theorem A.4. Let a RLCFG of size g_{rl} generate (only) $T[1\,..\,n]$. Then there exists a data structure of size $O(g_{rl})$ that finds the occ occurrences in T of any pattern $P[1\,..\,m]$ in time $O(m \log n + occ \log^{\epsilon} n)$ for any constant $\epsilon > 0$.

This result is essentially obtained in our Section 6. In that section we use a specific RLCFG that allows us obtain a better complexity. However, in a general RLCFG, where

we must search for all the $\tau = m - 1$ possible splits of P, the application of Lemma 6.5 with complexities $f_e(\ell) = O(\ell)$ (Theorem A.2) and $f_h(\ell) = O(\log n)$ (Theorem A.3) yields $O(m \log n)$ time to find all the $m - 1$ ranges $[x_1 \,..\, x_2] \times [y_1 \,..\, y_2]$ to search for in the grid.

Combining that result with the linear-space grid representation and the mechanism to track the secondary occurrences on the grammar tree of a RLCFG described in Section 6, the result follows immediately.

A.5. Counting pattern occurrences

While we cannot generalize our result of Section 7 to arbitrary RLCFGs, our developments let us improve the best current result on arbitrary CFGs [Navarro 2019].

Theorem A.5. Let a CFG of size g generate (only) $T[1 \,..\, n]$. Then there exists a data structure of size $O(g)$ that computes the number of occurrences in T of any pattern $P[1 \,..\, m]$ in time $O(m \log^{2+\epsilon} n)$ for any constant $\epsilon > 0$.

Navarro [2019, Thm. 4] showed that the number of times $P[1 \,..\, m]$ occurs in $T[1 \,..\, n]$ can be computed in time $O(m^2 + m \log^{2+\epsilon} n)$ within $O(g)$ space for any CFG of size g. As explained in Section 7, he uses the same grid of our Section 6 for the primary occurrences, but associates with each point the number of occurrences triggered by it (which depend only on the point). Then, a linear-space geometric structure [Chazelle 1988] sums all the numbers in a range in time $O(\log^{2+\epsilon} g)$. Adding over all the $m - 1$ partitions of P, and considering the $O(m^2)$ previous time to find all the ranges [Claude and Navarro 2012], the final complexity is obtained.

With Lemma 6.5, and given our new results in Theorems A.2 and A.3, we can now improve Navarro's result to $O(m \log^{2+\epsilon} n)$ because the $O(m^2)$ term becomes $O(m \log n)$. However, this holds only for CFGs. Run-length rules introduce significant challenges, in particular the number of secondary occurrences do not depend only on the points. We only could handle this issue for the specific RLCFG we use in Section 7. An interesting open problem is to generalize this solution to arbitrary RLCFGs.