Computing MEMs and Relatives on Repetitive Text Collections

Gonzalo Navarro, CeBiB and University of Chile, Chile

We consider the problem of computing the Maximal Exact Matches (MEMs) of a given pattern $P[1..m]$ on a large repetitive text collection $T[1..n]$ over an alphabet of size σ, which is represented as a (hopefully much smaller) run-length context-free grammar of size g_{rl}. We show that the problem can be solved in time $O(m^2 \log^\epsilon n)$, for any constant $\epsilon > 0$, on a data structure of size $O(g_{rl})$. Further, on a locally consistent grammar of size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$, the time decreases to $O(m \log m(\log m + \log^\epsilon n))$. The value δ is a function of the substring complexity of T and $\Omega(\delta \log \frac{n \log \sigma}{\delta \log n})$ is a tight lower bound on the compressibility of repetitive texts T, so our structure has optimal size in terms of n, σ, and δ. We extend our results to several related problems, such as finding k-MEMs, MUMs, rare MEMs, and applications.

Categories and Subject Descriptors: E.1 [**Data structures**]; E.2 [**Data storage representations**]; E.4 [**Coding and information theory**]: Data compaction and compression; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*

Additional Key Words and Phrases: Repetitive texts; Substring complexity; Grammar compression; Locally consistent parsing; Compressed data structures; MEMs; MUMs

1. INTRODUCTION AND RELATED WORK

Inexact sequence matching is the norm in Bioinformatic applications. Mutations in the genomes, and even the possible errors that arise in the sequence acquisition process, makes researchers expect to see differences between the patterns they look for and what they call their occurrences [Gusfield 1997; Ohlebusch 2013; Mäkinen et al. 2015]. For example, when assembling a genome one must align the reads (which are sequences obtained from the DNA of an individual, a few hundreds or thousands nucleotides long) to a reference genome (of length typically in the billions) or a set thereof. For the reasons above, the read may not appear in the genome in exact form, so one looks for the longest substrings of the read that appear in the genome, to determine the most likely positions where to align it. In pangenomics, one may compare the substrings of a whole gene or chromosome against another, or against a set of genomes representative of a population, to find conserved regions and spot the places that differ, which may indicate genetic variations or diseases.

Those examples are applications of one of the most relevant tools for inexact matching, which is finding the Maximal Exact Matches (MEMs) of a given pattern $P[1..m]$ in a text $T[1..n]$. A MEM is a maximal substring $P[i..j]$ that appears in T (i.e., $P[i-1..j]$ and $P[i..j+1]$ are out of bounds or do not occur in T). In the alignment of reads, m is in the hundreds or thousands; when aligning a chromosome m can surpass the millions.

Supported by Fondecyt grant 1-200038, Chile, and Basal Funds FB0001, ANID, Chile.
A preliminary version of this article appeared in *Proc. CPM'23* [Navarro 2023].
Author's addresses: Gonzalo Navarro, Center for Biotechnology and Bioengineering (CeBiB) and Department of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl

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

If the text is one genome, n can be in the billions, but genomic collections may contain hundreds to thousands of genomes.

In this paper we are interested in the case where T is known in advance and thus can be indexed so that, later, we can efficiently find the MEMs of many patterns P in it. This is the case in the two applications we have mentioned; in the pangenomic case T can be taken as the concatenation of all the genomes in the collection. Many other applications of the problem of finding the MEMs of a new string in a static collection of strings are mentioned by Gusfield [1997], Ohlebusch [2013], and Mäkinen et al. [2015], including simplified problems like finding longest common substrings and equivalent problems like computing matching statistics.

Finding MEMs on an indexed text T is a classic problem in stringology and can be solved in optimal $O(m)$ time using a suffix tree of T [Weiner 1973; McCreight 1976]. Gusfield [1997, Sec. 7.8], for example, shows how to solve the analogous problem of computing matching statistics (we discuss this equivalence later). In applications where we handle massive texts T, however, suffix trees are too large to be maintained in main memory, even if they use linear space. The suffix tree of a single human genome, for example, whose length is about 3 billion bases, may require 60GB of memory with a decent implementation. This rules out suffix trees to represent genomes on the large bioinformatic collections that are arising, and make researchers look for alternatives using less space. For example, Ohlebusch et al. [2010] and Belazzougui et al. [2013] use indices based on the Burrows-Wheeler Transform (BWT) [Burrows and Wheeler 1994] to compute MEMs in time $O(m \log \sigma)$, where σ is the alphabet size. Such indices take just a few GBs on a human genome, an order of magnitude below the space required with suffix trees.

When considering large collections of genomes, however, even such sharp space reduction can be insufficient. Projects to sequence 100,000 human genomes have been completed[1], and current projects aim to sequencing millions of human genomes[2]. In that scenario, even the BWT-based representations will need petabytes of main memory to run, or resort to orders of magnitude slower disk storage.

A fortunate situation is that many of the fastest growing text collections, including genome collections, are highly repetitive [Navarro 2021a]: two genomes of the same species feature a small percentage of differences only. Several text indices exploiting repetitiveness to reduce space have appeared [Navarro 2021b]. While probably not competitive with the BWT-based indices we mentioned when indexing one genome, those indices may take orders of magnitude less space than the raw data, not just of its indices, when representing a collection of many genomes.

Those compressed indices support exact pattern matching, that is, they can list all the positions where P occurs in T. While useful, this is insufficient to efficiently implement the MEM finding algorithms. This is the problem we address in this paper.

1.1. MEM finding with indices for repetitive text collections

The classic MEM-finding algorithm runs on a suffix tree, and those that run on BWT-based data structures emulate it. Compressed suffix trees for highly repetitive text collections do exist, but do not compress that much. Gagie et al. [2020] show how to simulate a suffix tree within space $O(r \log \frac{n}{r})$, where r is the number of equal-letter runs in the BWT of T. This representation can simulate the suffix-tree-based algorithm in time $O(m \log \frac{n}{r})$ if we run it backwards on P, using operations *parent* and *Weiner link* instead of *child* and *suffix link*. The problem is the space: while r is an accepted measure of repetitiveness [Kempa and Kociumaka 2020], it is a weak one [Navarro

[1] `https://www.genomicsengland.co.uk/initiatives/100000-genomes-project`
[2] `https://b1mg-project.eu`

2021a; Kempa and Kociumaka 2020], and multiplying it by $\log \frac{n}{r}$ makes it grow by an order of magnitude. Current implementations of compressed suffix trees for repetitive texts achieve remarkable space, but still use at least 2–4 bits per symbol [Russo et al. 2011; Farruggia et al. 2018; Cáceres and Navarro 2022; Boucher et al. 2021a].

Another trend has been to expand the functionality of a more basic compressed text index for repetitive text (typically, a compressed suffix array [Manber and Myers 1993]) so as to support specific operations, MEMs in our case. Bannai et al. [2020] show how to compute matching statistics (from where MEMs are easily extracted in $O(m)$ time) by extending the RLBWT-index [Mäkinen et al. 2010], in $O(m(s + \log\log n))$ time and $O(r)$ space, with the help of a data structure that provides access to a symbol of T in time $O(s)$. This can be, for example, the samples of the RLBWT-index, which add $O(n/s)$ space to the index, or a context-free grammar of T, which provides access in time $s = O(\log n)$ [Bille et al. 2015]. Various implementations of this idea [Rossi et al. 2022; Boucher et al. 2021b; Tatarnikov et al. 2023] showed its practicality on large genome collections, with indices that are an order of magnitude smaller than the text.

All those results have been obtained on the so-called *suffix-based* compressed indices for repetitive collections [Navarro 2021b]. This is natural because those emulate variants of suffix trees or arrays, which simplifies the problem of simulating the suffix tree traversal of the classic MEM-finding algorithm. Even a naive search for all the $O(m^2)$ substrings of P can be run in $O(m^2 \log\log n)$ time on those $O(r)$-sized indices.

The problem is much harder on the so-called *parsing-based* indices [Navarro 2021b]. Those are potentially smaller than the suffix-based indices because they build on stronger measures of repetitiveness. For example, the size g of the smallest context-free grammar that generates T is usually considerably smaller than r [Navarro 2021a]. Because these indices cut T into phrases, even exact pattern matching is complicated because the occurrences of P can appear in many different forms, and many possible cuts of P must be tried out ($m-1$ in the general case) [Claude et al. 2021]. This makes the problem of finding MEMs considerably harder. We are only aware of the results of Gao [2022], who computes matching statistics in time $O(m^2 \log^\epsilon \gamma + m \log n)$ using $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space (for any constant $\epsilon > 0$), or $O(m^2 + m \log \gamma \log\log \gamma + m \log n)$ time using $O(\gamma \log \gamma)$ further space. Here $\delta \le \gamma$ are lower-bounding measures of repetitiveness [Kempa and Prezza 2018; Christiansen et al. 2020]. The size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ matches a tight lower bound on the size of compressed representations of T [Kociumaka et al. 2023], so a structure of this size uses asymptotically optimal space for every n, σ, δ.

1.2. Our contribution

MEM finding is a fundamental bridge between exact and inexact pattern matching. Inexact matching is essential in many applications, most prominently bioinformatic ones, that feature huge yet repetitive sequence collections that render standard solutions inapplicable. Our paper provides the first bridge to inexact matching within optimal δ-bounded space and subquadratic time, thereby offering manageable space and time simultaneously. We now provide the details.

Let g_{rl} be the size of any run-length context-free grammar generating T (those include and extend classic context-free grammars). The smallest such grammar is of size $g_{rl} = O(\delta \log \frac{n \log \sigma}{\delta \log n})$ [Kociumaka et al. 2023]. We first show that, on an index of size $O(g_{rl})$, one can compute the MEMs in time $O(m^2 \log^\epsilon g_{rl})$, for any constant $\epsilon > 0$. This is done by sliding the window $P[i\,..\,j]$ of the classic algorithm while we simulate the process of searching for that window with the grammar. The simulation is carefully crafted to avoid expensive operations, so the time stays proportional to the number of cuts tried out on a single search for P. The space $O(g_{rl})$ is the least known to support

direct access to T with logarithmic time guarantees [Navarro 2021a], so improving our space is likely to involve breaking this long-standing barrier as well. Our result essentially matches the first one of Gao [2022], which, although he did not claim so, could also run within $O(g_{rl})$ space.

We further show that, on a particular grammar featuring local consistency properties [Kociumaka et al. 2024], we can reduce the time to $O(m \log m(\log m + \log^\epsilon n))$ by exploiting the fact that only $O(\log(j-i+1))$ cuts need to be tried out for $P[i\,..\,j]$, and using more sophisticated techniques to amortize the costs. This grammar is of size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$, which is optimal for every n, σ, and δ, and within this space we sharply break the quadratic time of previous solutions that run in grammar-bounded space.

We note that a recent result [Kempa and Kociumaka 2023] (simultaneous with our conference version [Navarro 2023]) offers suffix-based functionality within parsing-based space, by computing suffix array entries within $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space. While this result is remarkable in its generality and could be used to find MEMs, the time to compute suffix array entries is $O(\log^{4+\epsilon} n)$, which would render a MEM-finding solution based on it not competitive with our results.

We then turn to consider several relatives of the MEM finding problem, adapting our main algorithm to solve them:

— A natural generalization of MEMs are k-MEMs, the maximal substrings of P that appear at least k times in T. Those identify the parts of P that have sufficient support in T, for example regions of a gene that appear in most genomes of a collection, or regions in a reference genome that have sufficient coverage in a set of reads. Even with k given at query time, this problem is also easily solved in $O(m)$ time with a suffix tree [Navarro 2016], but obtaining the same on grammars is not so direct. We generalize our results on MEMs to find k-MEMs in time $O(km^2 \log^\epsilon n)$ within $O(g_{rl})$ space, or in time $O(m \log m(\log m + k \log^\epsilon n))$ within $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space. For $k = \omega(\log^2 n)$, we provide faster solutions that run in time $O(m^2 \log^{2+\epsilon} n)$ and $O(g)$ space, or in time $O(m \log m \log^{2+\epsilon} n)$ and $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space.
— A stricter version of MEMs are MUMs (maximal unique matches), which are maximal matches that appear exactly once in P and in T. MUMs have various applications to sequence alignment [Delcher et al. 1999; Mäkinen et al. 2015; Giuliani et al. 2022]. Classical solutions using suffix trees [Sung 2010] and suffix arrays [Ohlebusch 2013] compute MUMs in $O(m+n)$ time, though they are easily seen to take $O(m)$ time if P and T are indexed separately. MUMs are also computed in time $O(m \log \sigma)$ using a BWT-based index [Belazzougui et al. 2013]. The only compressed-space solution for repetitive texts we know [Giuliani et al. 2022] computes MUMs in $O(r)$ space (plus a grammar on T) and $O(m \log n)$ time. We compute MUMs within the same space and time complexities as for finding the MEMs.
— A natural generalization of MUMs are rare MEMs [Ohlebusch and Kurtz 2008], which also have applications in whole genome alignment [Ohlebusch 2013, p. 419]. We say that a MEM is k-rare if it appears at most k times in P and in T, so MUMs are 1-rare MEMs. We show that k-rare MEMs can be computed within the same space and time of k-MEMs.

We finally show, through applications to problems like data compression and genome assembly, that our techniques open the door to using parsing-based indices in stringology problems that had been addressed only through the more powerful (but more space-consuming) suffix-based ones.

Compared to the conference version of this paper [Navarro 2023], the key result, Section 6, was largely rewritten due to simplifications, improvements, and filling con-

siderable gaps of the conference version. Several minor problems were fixed in other sections as well. A new set of extensions of the basic result to related problems, as well as applications of our result, are also included in Sections 7 and 8. Finally, the whole paper includes more detailed explanations, figures, and a better presentation.

The roadmap of the paper is as follows. In Section 2 we formally define MEMs, the problem of finding them, and the variants we consider in the paper. We also describe the algorithms for finding MEMs and how they are adapted to find their variants. In Section 3 we describe the basic grammar-based index for pattern matching, on which our simple solution builds. This simple solution, quadratic in m, is described in Section 4. Section 5 then describes the more sophisticated locally consistent grammar we will use, and how the pattern matching problem is solved on it. The more complex, subquadratic, algorithm to find MEMs on locally consistent grammars is then described in Section 6. This is the central result of the article. Section 7 shows how the tools we have developed can be used to solve the related problems we have considered: k-MEMs, MUMs, k-rare MEMs, and so on. We explore some direct and not so direct applications of our results in Section 8. We conclude and give future work directions in Section 9.

2. MAXIMAL EXACT MATCHES (MEMS) AND RELATIVES

We use the classic notation on strings $S[1..n]$, so $S[i]$ is the ith symbol of S, $S[i..j]$ denotes $S[i]\cdots S[j]$ (i.e., the concatenation of symbols $S[i]$ to $S[j]$ and the empty string ε if $i > j$), $S[..j] = S[1..j]$ and $S[i..] = S[i..n]$. The concatenation of strings S and S' is denoted $S \cdot S'$. We assume that the reader is familiar with the concepts related to suffix trees [Weiner 1973; McCreight 1976; Crochemore and Rytter 2002]. In particular, we will use the RAM model of computation and assume the alphabet is integer, which enables suffix trees where one can move from a node to a child, by a given alphabet symbol, in constant time. Those suffix trees can be built in linear time [Farach-Colton et al. 2000], plus linear expected time to build perfect hash functions on the children of every node.

We start by defining the problems of finding MEMs and k-MEMs, and how to solve them on suffix trees.

Definition 2.1. A *Maximal Exact Match (MEM)* of a pattern $P[1..m]$ in a string T is a nonempty substring $P[i..j]$ that occurs in T, but in addition

— $i = 1$ or $P[i-1..j]$ does not occur in T, and
— $j = m$ or $P[i..j+1]$ does not occur in T.

Definition 2.2. A *k-MEM* of a pattern $P[1..m]$ in a string T is a nonempty substring $P[i..j]$ that occurs at least k times in T, but in addition

— $i = 1$ or $P[i-1..j]$ occurs fewer than k times in T, and
— $j = m$ or $P[i..j+1]$ occurs fewer than k times in T.

Definition 2.3. Given a text $T[1..n]$ that can be preprocessed, the *MEM-finding problem* is that of, given a pattern $P[1..m]$, return the range (i, j) of each of its MEMs $P[i..j]$ in T, in increasing order of i (and j). A position where each MEM occurs in T must also be returned. The *k-MEM finding problem* is defined analogously, for k given at query time.

The MEM and k-MEM finding problems can be solved in $O(m)$ time with a suffix tree of T. Algorithm 1 shows the solution for MEMs, abstracting away some complications of implementing it on the long edges of suffix trees.

The algorithm slides a window $P[i..j]$ of variable size along P, maintaining the invariants that (i) every MEM ending before j has already been reported, (ii) the longest suffix of $P[..j]$ that occurs in T is $P[i..j]$, and (iii) v is the suffix tree locus of $P[i..j]$. It

```
1  i ← 1; j ← 0 ;
2  v ← suffix tree root ;
3  while j < m do
4  |  if v has no child labeled P[i] then
5  |  |  i ← i + 1 ; j ← j + 1 ;
6  |  end
7  |  else
8  |  |  while j < m and v has a child labeled P[j + 1] do
9  |  |  |  j ← j + 1 ; v ← the child of v by P[j + 1] ;
10 |  |  end
11 |  |  report (i, j) with some occurrence of v ;
12 |  |  while i ≤ j < m and v has no child labeled P[j + 1] do
13 |  |  |  i ← i + 1 ; v ← the suffix link of v ;
14 |  |  end
15 |  end
16 end
```

Algorithm 1: Finding the MEMs of $P[1\,..\,m]$ in T using the suffix tree of T.

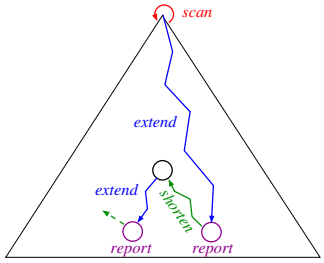

Fig. 1. Schematic view of the phases of the MEM finding algorithm on a suffix tree.

iterates along four phases: scanning, extending, reporting, and shortening; see Fig. 1. The scanning phase (lines 4–6) considers the case $i = j + 1$ (so v is the suffix tree root), where i and j are increased as long as one cannot descend from v by $P[j + 1]$. The extending phase (lines 8–10) starts when we have found a child of the root by $P[i]$ and then descend from v as much as possible by $P[i\,..\,j]$, increasing j. When we cannot further descend from the locus of $P[i\,..\,j]$ by $P[j + 1]$, the invariants imply that $P[i\,..\,j]$ is a MEM, which is reported in line 11. Finally, the shortening phase (lines 12–14) establishes the conditions that will allow increasing j again, by increasing i until $P[i\,..\,j+1]$ occurs in T (or $i > j$). The tree suffix link is used to move from the locus of $P[i\,..\,j]$ to that of $P[i + 1\,..\,j]$. Since each step increases i or j, the total time is $O(m)$.

To adapt the algorithm to k-MEMs, we just ask that the suffix tree nodes we reach by $P[j + 1]$ not only exist, but also that their subtrees contain k leaves at least.

Other notions related to MEMs are the MUMs and the k-rare MEMs.

Definition 2.4. A *k-rare MEM* of a pattern $P[1\,..\,m]$ in a string T is a nonempty substring $P[i\,..\,j]$ that occurs in both P and T, at most k times in both cases. Further,

— $i = 1$ or $P[i - 1\,..\,j]$ does not occur in T, and
— $j = m$ or $P[i\,..\,j + 1]$ does not occur in T.

A 1-rare MEM is called a *Maximal Unique Match (MUM)*.

The problem of finding the k-rare MEMs of T is defined analogously as that of finding the k-MEMs. A variant of Algorithm 1 to find the k-rare MEMs uses also a suffix tree of P, and traverses both suffix trees in synchronization, maintaining the locus of $P[i\,..\,j]$ in both suffix trees as it slides the window $P[i\,..\,j]$. It reports (i, j) in line 11 only if the number of leaves descending from both loci is at most k. The time is still $O(m)$, now in expectation due to the need of building the suffix tree of P.

3. GRAMMAR BASED INDICES

Let $T[1\,..\,n]$ be a text. Grammar-based compression of T consists in replacing it by a context-free grammar (CFG) that generates only T [Kieffer and Yang 2000]. The compression ratio is then the size of the grammar divided by the text size.

We consider a slightly more powerful type of grammar called run-length CFG (RL-CFG), which includes run-length rules of constant size. We disallow rules of the form $A \to \varepsilon$, which are easily removed without increasing the grammar size.

Definition 3.1. A *Run-Length Context-Free Grammar (RLCFG)* for T is a context-free grammar that generates only T, having exactly one rule per nonterminal A. The rules are of the form $A \to B_1 \cdots B_t$ for $t > 0$ and terminals or nonterminals B_i (this rule is said to be of size t), and of the form $A \to B^t$ for $t > 1$ and a terminal or nonterminal B, which is identical to $A \to B \cdots B$ with t copies of B, but is said to be of size 2. The *size* of the RLCFG is the sum of the sizes of its rules. A *Context-Free Grammar (CFG)* for T is a RLCFG for T that does not use rules of the form $A \to B^t$.

Clearly, the size g_{rl} of the smallest RLCFG for T is always less than or equal to the size g of the smallest CFG for T. Grammar-based compression (with or without run-length rules) has proved to be particularly effective on highly repetitive texts [Navarro 2021b]. While finding the smallest grammar is NP-hard [Charikar et al. 2005], heuristics like RePair obtain very good results [Larsson and Moffat 2000].

3.1. The grammar tree

Since they have exactly one rule per nonterminal, the RLCFGs that generate a single string T have a unique parse tree, defined as follows [Christiansen et al. 2020, Sec. 4].

Definition 3.2. The *parse tree* of a RLCFG for T has a root labeled with the initial symbol. If a node is labeled A and its rule is $A \to B_1 \cdots B_t$, then the node has t children labeled $B_1, \ldots, B_t$ left to right. If its rule is $A \to B^t$, then the node has t children labeled B. The ith left-to-right leaf of the parse tree is thus labeled $T[i]$.

While the parse tree has size $\Omega(n)$, a convenient representation of a RLCFG is the so-called grammar tree, which is of size $O(g_{rl})$ [Christiansen et al. 2020, Sec. 6].

Definition 3.3. The *grammar tree* of a RLCFG is obtained by pruning its parse tree, preserving the leftmost internal node labeled A for each nonterminal A, and converting the others to leaves. Further, for the remaining internal nodes labeled A with rules $A \to B^t$ we preserve their first child only, replacing the other $t - 1$ children (which are leaves) with a single special leaf labeled $B^{[t-1]}$. If the RLCFG size is g_{rl}, its grammar tree has $g_{rl} + 1$ nodes.

We will sometimes identify a nonterminal with its (only) internal node in the grammar tree. Fig. 2 gives an example.

We call $exp(A)$ the string of terminals to which symbol A expands, that is, $exp(a) = a$ for terminals a, $exp(A) = exp(B_1) \cdots exp(B_t)$ if $A \to B_1 \cdots B_t$, and $exp(A) = exp(B)^t$ (i.e., t concatenations of $exp(B)$) if $A \to B^t$. The grammar tree defines a parse (or partition into substrings called phrases) of T, as follows.

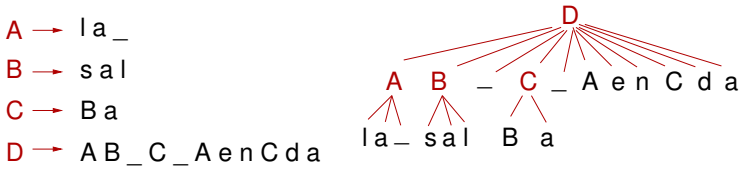

Fig. 2. A grammar for $T =$ la_sal_sala_la_ensalada and its grammar tree. For example, $exp(\mathsf{C}) =$ sala, and the parsing induced by the grammar is l · a · _ · s · a · l · _ · sal · a · _ · la_ · e · n · sala · d · a.

Definition 3.4. The grammar tree, with leaves $v_1, \ldots, v_k$, induces the *parse* $T = exp(v_1) \cdot exp(v_2) \cdots exp(v_k)$ into *phrases* $exp(v_i)$.

3.2. Primary occurrences

A classic grammar-based index [Claude et al. 2021] divides the occurrences of a pattern $P[1\,..\,m]$ into *primary* and *secondary*, depending on whether they cross a phrase boundary or lie within a phrase, respectively (if $m = 1$, its occurrences ending a phrase are taken as primary). The index builds on the fact that every pattern P has primary occurrences and that all the secondary ones can be found inside pruned leaves of nonterminals that contain other occurrences. The mechanism to find the primary occurrences is based on the parsing, but defined in a particular way to avoid reporting multiple times the primary occurrences that cross several phrase boundaries. The mechanism was extended to RLCFGs [Christiansen et al. 2020, Sec. 6 and App. A].

Definition 3.5. Let $\mathcal{X}$ and $\mathcal{Y}$ be multisets of strings defined as follows. For each rule $A \to B_1 \cdots B_t$, for each $1 < s \leq t$, the string $exp(B_{s-1})^{rev}$ (i.e., $exp(B_{s-1})$ read backwards) is inserted in $\mathcal{X}$ and the string $exp(B_s) \cdots exp(B_t)$ is inserted in $\mathcal{Y}$; we say those two are *corresponding* strings. Similarly, for each rule $A \to B^t$, $exp(B)^{rev}$ is inserted in $\mathcal{X}$ and $exp(B)^{t-1}$ is inserted in $\mathcal{Y}$. A grid $\mathcal{G}$ has one row per string in $\mathcal{Y}$ and one column per string in $\mathcal{X}$. After lexicographically sorting $\mathcal{X}$ and $\mathcal{Y}$, a point (x, y) is set in $\mathcal{G}$ if the xth string of $\mathcal{X}$ corresponds to the yth string of $\mathcal{Y}$.

The grammar-based index includes a Patricia tree $T_{\mathcal{X}}$ storing the strings of $\mathcal{X}$ and another Patricia tree $T_{\mathcal{Y}}$ storing the strings of $\mathcal{Y}$ [Morrison 1968]. We add some data to the Patricia tree nodes for convenience. Each Patricia tree node v stores the range $[v^1, v^2]$ of the left-to-right ranks of the leaves descending from v. The edges of the Patricia tree nodes can represent strings, so prefixes that end in the middle of an edge that leads to a node v correspond to *virtual* nodes u; the range $[u^1, u^2]$ is the same $[v^1, v^2]$. The nodes v also store their string depth $|v|$, which is also easily computed for virtual nodes as we descend or ascend in the Patricia tree. Finally, let $X = exp(B_{s-1})^{rev} \in \mathcal{X}$ be one string descending from node $x \in T_{\mathcal{X}}$, then we store $\langle x \rangle = B_{s-1}$ associated with x. Similarly, a node $y \in T_{\mathcal{Y}}$ that prefixes $exp(B_s) \cdots exp(B_t) \in \mathcal{Y}$ stores $\langle y \rangle = B_s$ (from where we can obtain the subsequent siblings $B_{s+1} \cdots$).

Each primary occurrence consists of a suffix of some string $X \in \mathcal{X}$ matching $P[1\,..\,r]$ corresponding to some string $Y \in \mathcal{Y}$ whose prefix matches $P[r+1\,..\,m]$, for some $1 \leq r < m$ (if $m = 1$, it is just a suffix of X matching P) [Christiansen et al. 2020, Sec. A.4]. Therefore, to find the primary occurrences of P, the index tries out every cutting point r, and searches $T_{\mathcal{X}}$ for $P[1\,..\,r]^{rev}$ and $T_{\mathcal{Y}}$ for $P[r+1\,..\,m]$. If both nodes $x \in T_{\mathcal{X}}$ and $y \in T_{\mathcal{Y}}$ exist, then the points in the orthogonal range $[x^1, x^2] \times [y^1, y^2]$ of $\mathcal{G}$ represent the primary occurrences of P cut at position r, and are efficiently found with a geometric data structure on $\mathcal{G}$. The actual index stores a pointer to the node B_{s-1} in the grammar tree; we instead store the position p of T where $exp(B_{s-1})$ ends for such point, so we know that P occurs in $T[p - r + 1\,..\,p - r + m]$. See Fig. 3.

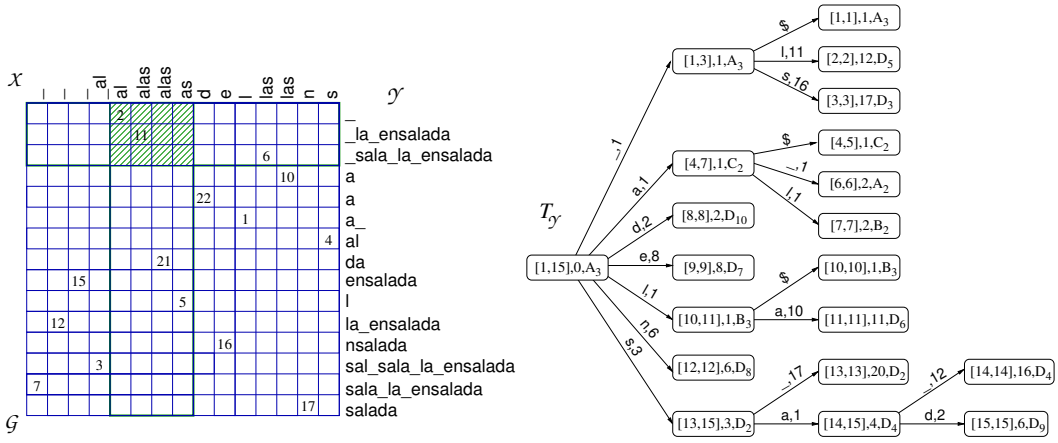

Fig. 3. The grid for the grammar of Fig. 2, and the ranges induced by the search for $P =$ "a_". We obtain the position 2, which is inside the internal node for nonterminal A of the grammar tree, and the position 11, which is inside the internal node for nonterminal D of the grammar tree (starting inside child C). The occurrence at position 14 is secondary. On the right, the Patricia tree $T_{\mathcal{Y}}$. We write $[y^1, y^2], |y|, \langle y \rangle$ inside every node y. We use \$ to denote the string terminator.

Since $|\mathcal{X}|, |\mathcal{Y}| \leq g_{rl}$, both the Patricia trees and the grid take $O(g_{rl})$ space. The index also needs to verify the matches of the Patricia trees. For this purpose it uses an $O(g_{rl})$-space data structure $\mathcal{E}$ that can extract, in $O(\ell)$ time, any length-ℓ prefix or suffix of $exp(A)$, for any nonterminal A [Christiansen et al. 2020, Lem 6.6]. If x is a node of $T_{\mathcal{X}}$, its corresponding string is the $|x|$-length reversed suffix of any string between the x^1th and the x^2th in $\mathcal{X}$. We can then obtain the string representing node x by using $\mathcal{E}$ on $\langle x \rangle$; analogously using $\mathcal{E}$ on $\langle y \rangle$ for nodes $y \in T_{\mathcal{Y}}$.

3.3. Secondary occurrences

Each primary occurrence found inside a nonterminal A triggers a number of secondary occurrences:

(1) Every other occurrence of A in the grammar tree (which is necessarily a leaf) contains an occurrence of P.
(2) The occurrence in A also occurs in the parent of A in the grammar tree.

From each primary occurrence in A, then, we recursively trigger searches to the next occurrence of A and to the parent of A in the grammar tree. If we keep track of the offset of the occurrence within $exp(B)$ for each node B in the way, we can report the position of a secondary occurrence each time we reach the root of the grammar tree. In the classic CFG-based index, this is shown to amortize to constant time per secondary occurrence if each nonterminal is the root or appears at least twice in the grammar (the grammar can be modified to enforce this condition) [Claude et al. 2021]. The mechanism was extended to RLCFGs and to cases where the grammar cannot be modified [Christiansen et al. 2020, Sec. 6 and A.4]: each node points to its closest ancestor that is the root or occurs twice in the grammar tree, and the run-length nodes are treated in a special form to extract all the secondary occurrences they encode.

A particularly relevant observation for this paper [Navarro 2019] is that, on CFGs, the number of occurrences triggered by a primary occurrence depends only on that primary occurrence, hence the number can be precomputed and associated with its point in the grid.

4. A QUADRATIC-TIME SOLUTION FOR MEMS

We now present a quadratic-time solution for finding MEMs that works with any RLCFG of size g_{rl} for T. We use the $O(g_{rl})$-space data structures described in the previous section. Since CFGs are also RLCFGs, our algorithm also runs on any CFG.

The generic idea follows that of Algorithm 1, sliding a window $P[i\,..\,j]$ along the pattern. A trivial solution would be to use the classic search described in Section 3 for every window $P[i\,..\,j]$, to determine whether it occurs in T. Applied on any CFG of size g, such solution takes time $O((j-i+1)^2 + (j-i+1)\log^\epsilon g)$ for any constant $\epsilon > 0$ if we resort to the classic index [Claude et al. 2021, Cor. 1], because it needs to check every cutting point inside $P[i\,..\,j]$. This leads to $O(m^3 + m^2 \log^\epsilon g)$ time complexity. Using the best current solution for arbitrary RLCFGs [Christiansen et al. 2020, Thm. A.4], this search takes $O((j-i+1)\log n)$ time, leading to $O(m^2 \log n)$ total time for finding the MEMs of P in T within $O(g_{rl})$ space. In this section we will reduce this time only slightly, but will set the grounds of a much better solution in the next section.

The key aspect of the solution is to avoid redoing the search for every window. In particular, the trivial solution computes all the cuting points for every window $P[i\,..\,j]$, while two consecutive windows share most of them. We instead maintain, along the process, a set of so-called *active positions* $r \in [i\,..\,j]$.

Definition 4.1. A position $r \in [i\,..\,j]$ is *active* on the window $P[i\,..\,j]$ iff there is a primary occurrence of $P[r\,..\,j]$ in T with the cut $P[r] \cdot P[r+1\,..\,j]$.

Note that, since we slide the window $P[i\,..\,j]$ forwards, once a position r becomes inactive, it will not become active again. Note also that it is possible that $r = j$ and thus $P[r+1\,..\,j] = \varepsilon$, which is needed to find the MEMs of length 1.

4.1. Algorithm

The algorithm maintains invariants (i–ii) of Algorithm 1, that is, when the window is $P[i\,..\,j]$, all MEMs ending before j have already been reported and the longest suffix of $P[..\,j]$ that occurs in T is $P[i\,..\,j]$. Instead of the locus of $P[i\,..\,j]$ maintained in Algorithm 1, our algorithm maintains the set $R \subseteq [i\,..\,j]$ of active positions for $P[i\,..\,j]$, and for each such active position $r \in R$ it stores (see the left of Fig. 4):

— The node $y_r \in T_{\mathcal{Y}}$ corresponding to $P[r+1\,..\,j]$; this node can be virtual. Note that $[y_r^1, y_r^2]$ is the same range of rows in $\mathcal{G}$ of the strings of $\mathcal{Y}$ that start with $P[r+1\,..\,j]$.
— The length ℓ_r of the maximum prefix of $P[r+1\,..]$ that prefixes a string in $T_{\mathcal{Y}}$; note that r is active iff $r + \ell_r \geq j$, so ℓ_r can be zero if $r = j$.
— The node $x_r \in T_{\mathcal{X}}$ corresponding to the longest prefix of $P[i\,..\,r]^{rev}$ that exists in $T_{\mathcal{X}}$, and such that there are points in $\mathcal{G}$ in the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$, that is, there is a primary occurrence in T with the cut $P[r-|x_r|+1\,..\,r] \cdot P[r+1\,..\,j]$. Note again that x_r can be virtual and that $[x_r^1, x_r^2]$ is the same range of columns in $\mathcal{G}$ of the strings of $\mathcal{X}$ that start with $P[r-|x_r|+1\,..\,r]^{rev}$. Further, note we are interested only in values $r - |x_r| + 1 \geq i$ because, by invariant (ii), $P[i-1\,..\,j]$ does not occur in T.

Our algorithm, depicted in Algorithm 2, iterates over j, from 0 to $m-1$, and at each cycle it extends the current window so that it ends in $j+1$. When $i = j+1$ (including when we start with $i = 1$ and $j = 0$), the window is empty and there are no active positions. This corresponds to the scanning phase of Algorithm 1; the other phases are not as clearly separated. Line 3 first sees if we can descend from the root of $T_{\mathcal{X}}$ by $P[j+1]$, so as to start a new nonempty substring $P[j+1, j+1]$. If this is not possible, it just increases i and proceeds with the next value of j. Otherwise, there will be active positions for the window ending at $j+1$ and we enter into the main process.

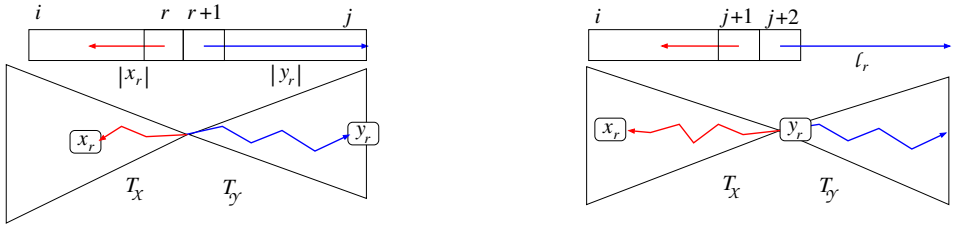

Fig. 4. On the left, the general relation between the nodes x_r and y_r for a general $r \in [i\,..\,j]$: while $|x_r|$ must be at most $r-i+1$ but it can be less, $|y_r|$ in $T_{\mathcal{Y}}$ must be exactly $j-r$; otherwise r becomes inactive. On the right, the computation we carry out for a new $r=j+1$: we compute ℓ_r, though y_r corresponds to the root of $T_{\mathcal{Y}}$, and compute x_r. Later, as j grows, y_r will descend in $T_{\mathcal{Y}}$ and x_r will possibly ascend in $T_{\mathcal{X}}$.

```
1  i ← 1; R ← ∅ ;
2  for j ← 0, ..., m − 1 do
3      if i = j + 1 and the root of T_X has no child labeled P[j + 1] then i ← i + 1 ;
4      else
5          y_{j+1} ← root of T_Y ;
6          v ← descend in T_Y as much as possible with P[j + 2 ..] ; ℓ_{j+1} ← |v| ;
7          x_{j+1} ← descend in T_X as much as possible with P[i .. j + 1]^rev ;
8          r_min ← j + 1 ;
9          for r ∈ R do
10             if r + ℓ_r ≤ j then R ← R \ {r} ;
11             else
12                 y_r ← child of y_r by P[j + 1] ;
13                 while range [x_r^1, x_r^2] × [y_r^1, y_r^2] is empty do
                       x_r ← (virtual) parent of x_r ;
14                 if |x_r| = 0 then R ← R \ {r} ;
15                 else if r − |x_r| < r_min − |x_{r_min}| then r_min ← r ;
16             end
17         end
18         l ← r_min − |x_{r_min}| + 1 ;
19         if l > i then
20             report (i, j) with position T[p − j + i .. p] ;
21             i ← l
22         end
23         if i ≤ j + 1 then
24             p ← j + 1 − r_min + some text position in [x^1_{r_min}, x^2_{r_min}] × [y^1_{r_min}, y^2_{r_min}] ;
25             R ← R ∪ {j + 1}
26         end
27     end
28 end
29 if i ≤ m then report (i, m) with position T[p − m + i .. p] ;
```

Algorithm 2: Finding the MEMs of $P[1\,..\,m]$ in T using a grammar-based index.

Lines 5–7 first create the new active position $r=j+1$, with the corresponding y_r set at the root of $T_{\mathcal{Y}}$. To compute ℓ_r, we descend in $T_{\mathcal{Y}}$ as much as possible by $P[r+1\,..]$. To compute x_r, we also descend in $T_{\mathcal{X}}$ as much as possible by $P[i\,..\,r]^{rev}$. Those are classic Patricia tree searches, first reaching a candidate node v by comparing only the

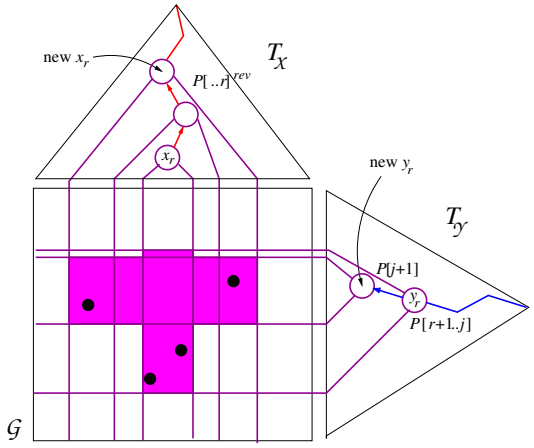

Fig. 5. Recomputing x_r in $T_{\mathcal{X}}$ when y_r descends in $T_{\mathcal{Y}}$ by $P[j+1]$. If the range in $\mathcal{G}$ becomes empty, we move to the parent of x_r (twice in this example) until it becomes nonempty again.

branching characters in the trie, and then verifying which ancestor of v is the correct answer. The verification proceeds by extracting the needed prefix from $\langle v \rangle$ in $T_{\mathcal{Y}}$ (at most $\ell_r + 1$ characters) or the needed suffix in $T_{\mathcal{X}}$ (at most $|x_r| + 1$ characters). See the right of Fig. 4.

Note that, once we compute ℓ_r, we know for every later j that we can descend from y_r by $P[j+1]$ iff $r + \ell_r \geq j + 1$, and therefore we can compute the child node on the Patricia tree without accessing the text, both for explicit and virtual nodes y_r. Thus, by computing ℓ_r once when the active position r is created, in time $O(\ell_r)$, we save all the accesses to T we would need to descend from virtual nodes $y_r \in T_{\mathcal{Y}}$. This would have been problematic because, when y_r is not the root, its text position is not phrase-aligned, and thus we cannot access its first symbols in constant time using $\mathcal{E}$ (recall Section 3.2).

Lines 8–17 then remove the positions that are no longer active and update the variables of the surviving ones. Line 10 first removes the active positions r where $r+\ell_r = j$. On the remaining ones, each y_r moves to its child by $P[j+1]$ in $T_{\mathcal{Y}}$ in line 12, without accessing T as explained.

Since moving down with y_r may shrink the interval $[y_r^1, y_r^2]$, line 13 updates the node x_r accordingly, because the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ could now become empty. For every active position r, as long as there are no points in $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$, we move x_r to its parent in $T_{\mathcal{X}}$. This process eventually terminates because, when x_r is the root and $[x_r^1, x_r^2]$ is the whole range of columns, we know that there are points in the band $[y_r^1, y_r^2]$ because it corresponds to the node y_r. Fig. 5 illustrates the process. If x_r became the root, however, r is not active anymore per Definition 4.1, so line 14 removes it from R.

Lines 8, 15, and 18 compute the active position $r_{\min}$ that yields occurrences starting at the leftmost position in the window, $l = r_{\min} - |x_{r_{\min}}| + 1$. This minimization includes the active position $j+1$ we have not yet inserted in R. It is then necessary to make i grow to l to re-establish invariant (ii) for $P[i\,..\,j+1]$. If $l > i$, then $P[i\,..\,j+1]$ does not occur in T and thus (i, j) was a MEM, by invariant (ii). Lines 20–21 then report the MEM (i, j) with its text position p (collected in the previous cycle of j) and increase i to l, since only $P[l\,..\,j+1]$ occurs in T. Increasing i could make it exceed $j+1$ and thus make the window empty; otherwise line 24 collects the text position p of some point in $[x_{r_{\min}}^1, x_{r_{\min}}^2] \times [y_{r_{\min}}^1, y_{r_{\min}}^2]$, to be reported in case $(i, j+1)$ turns out later to be a MEM.

Finally, $j+1$ is inserted in R as an active position. Line 27 reports the final MEM when j reaches m, which again is correct by invariant (ii).

The geometric queries performed on $\mathcal{G}$ in line 13 are called (orthogonal) *range emptiness queries*. These tell whether there is any point in a given rectangle of the grid. The other geometric queries are those of line 24, where we ask, in case a rectangle is not empty, for some point within it (and we extract the text position p from the satellite data of the point). This second query is generally a subproduct of the first, so both can be seen as the same query.

4.2. Analysis

For each value of j, Algorithm 2 spends $O(1)$ time per active position. Since there are $O(m)$ active positions at any time, this amounts to $O(m^2)$ time.

The exceptions are lines 6, 7, and 13, which are better charged to each active position r, from its creation to its inactivation. When r is created, we spend $O(m)$ time to compute $\ell_r \leq m$ and x_r (since $|x_r| \leq m$). Later, we can decrease $|x_r|$ several times, performing one range emptiness query in $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ per decrement of $|x_r|$ (in fact we can go directly to the lowest physical ancestor of x_r rather than to its possibly virtual parent node, as otherwise the range $[x_r^1, x_r^2]$ will not change). Thus, we perform overall $O(m^2)$ emptiness queries, up to m per position r along its life. Keeping the variables associated with active positions allows us amortizing these costs along the process.

A two-dimensional *range predecessor query* on $\mathcal{G}$ can be answered in $O(\log^\epsilon g_{rl})$ time and $O(g_{rl})$ space for any constant $\epsilon > 0$ [Belazzougui and Puglisi 2016]. This query returns the point with largest y coordinate in $[x_r^1, x_r^2] \times [0, y_r^2]$. That point has y coordinate $\geq y_r^1$ iff the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ is nonempty, so we determine range emptiness and obtain a point in the range if it exists, all in time $O(\log^\epsilon g_{rl})$. The total cost of the MEM finding algorithm is then $O(m^2 \log^\epsilon g_{rl})$, dominated by the $O(m^2)$ geometric queries.

The grammar data structures we use are those of the classic solution [Claude et al. 2021] (we would not obtain better times in this section by using the more sophisticated index of Christiansen et al. [2020]). Most of the construction cost is dominated by that of sorting $\mathcal{X}$ and $\mathcal{Y}$ in order to build the Patricia trees. Those sets can be sorted in time $O(g_{rl} \log^2 n)$ time and $O(g_{rl} \log n)$ space [Claude et al. 2021, Sec. 3 & 5.1], or in $O(n)$ time and space using the suffix arrays of T and T^{rev} [Manber and Myers 1993; Kärkkäinen et al. 2006]. While those construction times have been given for CFGs, they also apply to RLCFGs because we only introduce two strings for the rules of the form $A \to B^t$, and use general sorting algorithms. The other main component of the construction cost is that of the data structures that support the geometric queries on $\mathcal{G}$, which can be built in time $O(g_{rl}\sqrt{\log g_{rl}})$ [Belazzougui and Puglisi 2016].

THEOREM 4.2. *Assume we have a RLCFG of size g_{rl} that generates only $T[1..n]$. Then, for any constant $\epsilon > 0$, we can build a data structure of size $O(g_{rl})$ that finds the MEMs of any given pattern $P[1..m]$ in time $O(m^2 \log^\epsilon g_{rl}) \subseteq O(m^2 \log^\epsilon n)$, returning an occurrence of each MEM. The data structure can be built in $O(g_{rl} \log^2 n)$ time and $O(g_{rl} \log n)$ space, or in $O(n + g_{rl}\sqrt{\log g_{rl}})$ time and $O(n)$ space. The query process uses $O(m)$ additional space on top of the data structure.*

If we use the smallest RLCFG that generates T, then it holds that $g_{rl} = O(\delta \log \frac{n \log \sigma}{\delta \log n})$ [Navarro 2021a], which matches a lower bound on the space needed to represent repetitive strings (see next section for details). With this bound, the time of the algorithm can be written as $O(m^2(\log^\epsilon \delta + \log\log n))$. On the other hand, by using a range emptiness data structure of size $O(g_{rl} \log\log g_{rl})$ [Chan et al. 2011], the index is slightly larger but we can find the MEMs in time $O(m^2 \log\log g_{rl})$. As far as we know, however, the construction time of this data structure is not established.

5. INDEXING LOCALLY CONSISTENT GRAMMARS (LCGS)

Before entering into the details of our more sophisticated solution for finding MEMs, we must introduce some new concepts. A *locally consistent grammar (LCG)* is a kind of RLCFG that guarantees that equal substrings of T are covered by similar subtrees of the parse tree, differing in $O(1)$ nonterminals at each level of both subtrees. LCGs have been used to produce grammar-based indices that find all the primary occurrences with only a logarithmic number of cuts in P, thereby obtaining exact pattern searches in time that grows only linearly with m [Christiansen et al. 2020; Kociumaka et al. 2024]. In this paper we make use of the latest result [Kociumaka et al. 2024]. We present a lighter informal description; see the original paper for full details.

Since LCGs are not necessarily the smallest RLCFGs that generate a given text T, we will use another goal for the space achieved by a LCG [Raskhodnikova et al. 2013; Christiansen et al. 2020].

Definition 5.1. Given a text $T[1\,..\,n]$ we define, for every $\ell > 0$, T_ℓ as the number of distinct length-ℓ substrings in T, and then

$$\delta \;=\; \delta(T) \;=\; \max\{T_\ell/\ell,\ \ell > 0\}.$$

The set of values T_ℓ is called the *substring complexity* of T. For (almost) any choice of text length n, alphabet size σ, and measure δ, there exists family of texts sharing those parameters where $\Omega(\delta \log \frac{n \log \sigma}{\delta \log n})$ space (measured in $\log(n)$-bit words) is necessary to distinguish any text T of the family from the other members [Kociumaka et al. 2023]. We then aim at using space $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ for a grammar (and for an index), as this is asymptotically optimal. We note that, for every particular T with parameters n, σ, and δ, there exists a RLCFG of size $g_{rl} = O(\delta \log \frac{n \log \sigma}{\delta \log n})$ generating T [Kociumaka et al. 2024], so the size of the smallest RLCFG is a finer lower bound than this δ-based measure. Indeed, there are text families where $g_{rl} = o(\delta \log \frac{n \log \sigma}{\delta \log n})$ (e.g., if $T_n = (ab)^n$, then $\delta = \sigma = 2$, $g_{rl} = O(1)$, and $\delta \log \frac{n \log \sigma}{\delta \log n} = \Theta(\log n)$). The index of Theorem 4.2 is then potentially smaller, yet slower, than the one we will develop in this section.

5.1. The grammar

We first define the grammar [Kociumaka et al. 2024, Sec. 3], which is produced level by level, for $O(\log n)$ levels. Let Σ be the alphabet of the text $T[1\,..\,n]$ and also the set of grammar terminals. Let S_k be the sequence of terminals and nonterminals forming level k of the grammar. Let $\ell_k = (4/3)^{\lceil k/2 \rceil - 1}$, and let $\mathcal{A}_k$ be the set of symbols A of the previous level $k-1$ such that $|exp(A)| \leq \ell_k$. Only the symbols in $\mathcal{A}_k$ can be grouped to form new nonterminals in level k; the others are said to be "paused" in that level.

Our string at level 0 is $S_0 = T$. To form the string S_1, we detect the maximal *runs* of (at least 2) equal consecutive symbols in S_0 that are in $\mathcal{A}_1 = \Sigma$. For each such run, say of t symbols $a \in \mathcal{A}_1$, we create the rule $A \to a^t$ and replace the run by the nonterminal A. The resulting sequence after all the runs have been replaced is $S_1 = rle_1(S_0)$, which contains terminals and nonterminals. To form level 2, we define a function π_2 that reorders at random the distinct symbols of S_1, and use it to define *blocks* in S_1. Each "local minimum" position $0 < i < |S_1|$ such that

$$\pi_2(S_1[i-1]) > \pi_2(S_1[i]) < \pi_2(S_1[i+1])$$

terminates a block. We also set ends of blocks at $|S_1|$ and before and after every symbol not in $\mathcal{A}_2$ (which is still Σ per the formula of ℓ_k, so the runs introduced in S_1 cannot yet be grouped). For each distinct resulting block $S_1[i\,..\,j]$ we create a new rule $A \to S_1[i\,..\,j]$ and replace every occurrence of the same block in S_1 by A. The resulting string

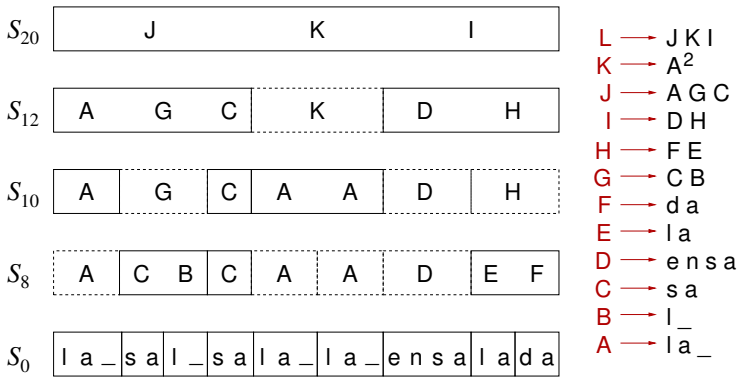

Fig. 6. Our LCG built over $T =$ "la_sal_sala_la_ensalada". We use the lexicographic values of terminals and nonterminals as their permutation, and show with rectangles the formed blocks (the underscore is taken as the smallest terminal). We show the levels S_k where the value of ℓ_k allows changes in the parsing; symbols that cannot yet be grouped are in dashed rectangles.

is called $S_2 = bc_2(S_1)$. The process continues in the same way for odd and even levels:

$$S_k = rle_k(S_{k-1}) \text{ if } k \text{ is odd},$$
$$S_k = bc_k(S_{k-1}) \text{ if } k \text{ is even},$$

until we reach $|S_k| = 1$ for some $k = O(\log n)$; see Fig. 6. The algorithm is Las Vegas type, trying out functions π_k until obtaining some desired grammar size; otherwise any functions π_k yield a correct grammar of size $O(n)$ and height $O(\log n)$. Kociumaka et al. [2024] prove that, after $O(1)$ attempts (i.e., in $O(n)$ expected time), a RLCFG of asymptotically optimal size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ is obtained.

A key property of this grammar is *local consistency*. Let $\mathcal{B}_k$ be the set of all the ends of level-k blocks:

$$\mathcal{B}_k = \{|exp(S_k[..j])|,\ 1 \le j \le |S_k|\},$$

where we are extending $exp(\cdot)$ homomorphically to strings. The cuts of level k that fall inside the substring at $T[i..j]$ have the following positions inside $T[i..j]$:

$$\mathcal{B}_k(i,j) = \{p - i + 1,\ p \in \mathcal{B}_k \cap [i..j-1]\}.$$

Local consistency makes the sets $\mathcal{B}_k(i,j)$ and $\mathcal{B}_k(i',j')$ similar if $T[i..j] = T[i'..j']$, except at the extremes: let $\alpha_k = \lceil 8\ell_k \rceil$, then $\mathcal{B}_k(i + 2\alpha_k, j - \alpha_k) = \mathcal{B}_k(i' + 2\alpha_k, j' - \alpha_k)$.

An additional property of the resulting grammar is that it is *locally balanced*: the subtree of the parse tree rooted at nonterminal A is of height $O(\log |exp(A)|)$. This is a consequence of the fact that in S_k there are fewer than $1 + 4(j - i + 1)/\ell_{k+1}$ blocks ending inside $T[i..j]$, and the height of A is never more than the level k of the string S_k where it was created.

5.2. Pattern searching

Let us now define which cuts of P we need to try out in order to capture all the primary occurrences with this grammar [Kociumaka et al. 2024, Sec. 4]. Since ends of blocks in $\mathcal{B}_k(i,j)$ correspond to the phrase endings where a primary occurrence $T[i..j] = P$ can be cut, our set of cutting positions must suffice to capture those possible block endings for all k and for every possible $T[i..j]$ that matches P. We define

$$M_k(i,j) = \mathcal{B}_k(i,j) \setminus [2\alpha_{k+1} + 1 .. j - i - \alpha_{k+1}]$$
$$\cup\ \{\min(\mathcal{B}_k(i,j) \cap [2\alpha_{k+1} + 1 .. j - i - \alpha_{k+1}])\},$$

that is, all the cutting points in the extremes, where the different occurrences of $T[i\,..\,j]$ may differ, and just the leftmost one in the part that is guaranteed to be equal. Over all the levels, we build

$$M(i,j) \;=\; \bigcup_{k\geq 0} M_k(i,j).$$

The key point [Kociumaka et al. 2024, Lem. 4.2] is that $M(i,j)$ depends only on the content of $T[i\,..\,j]$ (not on its position in T), so we can define $M(P) = M(i,j)$ if $P = T[i\,..\,j]$, and this is the same set for every possible occurrence of P in T. Further, $|M_k(i,j)| = O(1)$ and $|M(P)| = O(\log m)$. In operational terms, this means that, at query time, we parse P in $O(m)$ time using the same rules we defined for T, producing a parse tree of height $O(\log m)$ and finding the $O(\log m)$ cutting points $M(P)$.

6. FASTER MEM FINDING USING LCGS

The idea to use the index of the preceding section is to exploit the fact that $O(\log(j - i + 1))$ cutting points suffice to find all the primary occurrences of any window $P[i\,..\,j]$. We will then maintain the parse tree of $P[i\,..\,j]$, and the set $M(P[i\,..\,j])$, as we slide the window through P, and use them to maintain the number $|R|$ of active positions within $O(\log m)$. We also need more sophisticated mechanisms to avoid the quadratic costs in lines 6, 7, and 13 of Algorithm 2.

The new algorithm is shown in Algorithm 3. We first parse the pattern in a way analogous to the text in line 1, in $O(m \log\log m)$ time. We then compute in line 2 the deepest nodes vx_r and vy_r we can reach in $T_{\mathcal{X}}$ and $T_{\mathcal{Y}}$ by descending with $P[..\,r]^{rev}$ and $P[r+1\,..]$, respectively; note the depth ℓ_r of Algorithm 2 is now $|vy_r|$. These deepest nodes are computed for all $1 \leq r < m$ at once because this allows a more efficient computation, $O(m \log^2 m)$ time instead of quadratic.

In the main loop, lines 7–8 compute the set $M(i, j+1) = M(P[i\,..\,j+1])$ of $O(\log m)$ cutting points for $P[i\,..\,j+1]$, in $O(m \log m)$ overall time.

Lines 9–22 update R according to this new set. Lines 9–12 remove the active positions r that do not anymore belong to $M(i, j+1)$ or do not reach $P[j+1]$. The node y_r for the remaining positions is updated by descending in $T_{\mathcal{Y}}$; as in Algorithm 2, this does not need access to the text because the path towards vy_r has been verified. This takes $O(m \log m)$ time.

Lines 13–17 incorporate new active positions in R, which appeared in $M(i, j+1)$ and reach $P[j+1]$; now a position r can enter and leave $M(i,j)$, and hence R, more than once. Further, unlike in Algorithm 2, r may enter R not only when $r = j+1$, so y_r is not necessarily the root of $T_{\mathcal{Y}}$. Instead, y_r is now found as the highest ancestor of vy_r in $T_{\mathcal{Y}}$ with string depth $\geq j - r + 1$. This corresponds to a *weighted ancestor query*, where the weights of the nodes y are their string lengths $|y|$. This query can be solved in time $O(\log\log n)$, within $O(|T_{\mathcal{Y}}|)$ space and preprocessing time [Amir et al. 2007]. As we see in Section 6.2, this loop executes $O(m)$ times overall, so it contributes $O(m \log\log n)$ to the total time.

Finally, since all the ranges $[y_r^1, y_r^2]$ have potentially changed by including $P[j+1]$, lines 19–22 recompute for every $r \in R$ the lowest ancestor x_r of the deepest node vx_r such that $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ is nonempty. This is not done by successively going to the parent in $T_{\mathcal{X}}$ as in Algorithm 2, but using so-called (orthogonal) range successor queries. The total time of these queries along the whole process is $O(m \log m \log^{\epsilon} g)$, where $\epsilon > 0$ is any constant. The positions r such that their node x_r becomes the root of $T_{\mathcal{X}}$ are removed from R.

Lines 23–34 are as in Algorithm 2; again our range successor queries yield some point within the nonempty rectangle they find, for line 24. The total query time is then

Parse P, computing all P_k ; (Section 6.1)
Compute nodes $vx_r \in T_{\mathcal{X}}$ and $vy_r \in T_{\mathcal{Y}}$ for all $1 \leq r < m$; (Section 6.3)
$i \leftarrow 1$; $R \leftarrow \emptyset$;
for $j \leftarrow 0, \ldots, m-1$ **do**
  if $i = j+1$ and the root of $T_{\mathcal{X}}$ has no child labeled $P[j+1]$ **then** $i \leftarrow i+1$;
  else
    Advance all i_k/j_k until $P_k[i_k]/P_k[j_k]$ contain $P[i]/P[j+1]$; (Section 6.1)
    Compute $M(i, j+1)$; (Section 6.2)
    for $r \in R$ **do**
      if $r \notin M(i, j+1)$ **or** $r + |vy_r| \leq j$ **then** $R \leftarrow R \setminus \{r\}$;
      else $y_r \leftarrow$ child of y_r by $P[j+1]$;
    end
    for $r \in M(i, j+1) \setminus R$ (Section 6.2) **do**
      if $r + |vy_r| > j$ **then**
        $R \leftarrow R \cup \{r\}$;
        $y_r \leftarrow \text{weightedAncestor}(T_{\mathcal{Y}}, vy_r, j - r + 1)$;
      end
    end
    for $r \in R$ **do**
      $x_r \leftarrow \text{rangeExpand}([vx_r^1, vx_r^2], [y_r^1, y_r^2])$; (Section 6.4)
      if $|x_r| = 0$ **then** $R \leftarrow R \setminus \{r\}$;
    end
    $r_{\min} \leftarrow \text{argmin}\{r - |x_r|,\ r \in R\}$;
    $l \leftarrow r_{\min} - |x_{r_{\min}}| + 1$;
    if $l > i$ **then**
      report (i, j) with position $T[p - j + i \,..\, p]$;
      $i \leftarrow l$;
    end
    if $i \leq j+1$ **then**
      $p \leftarrow j + 1 - r_{\min}$ + some text position in $[x_{r_{\min}}^1, x_{r_{\min}}^2] \times [y_{r_{\min}}^1, y_{r_{\min}}^2]$
    end
  end
end
if $i \leq m$ **then report** (i, m) with position $T[p - m + i \,..\, p]$;

Algorithm 3: Finding the MEMs of $P[1 \,..\, m]$ in T using our LCG-based index.

$O(m(\log^2 m + \log\log n + \log m \log^\epsilon g)) \subseteq O(m \log m(\log m + \log^\epsilon n))$. We now describe all the algorithmic components of our solution.

6.1. Parsing the pattern

In this section we preprocess the pattern so that, later, we can efficiently have the parse tree of every substring $P[i \,..\, j]$ as we slide the window along P. We first parse the whole P, using the same permutations π_k used to parse T, so as to create sequences P_k analogous to those S_k built when parsing T. We note that $|P_k| = O(m/\ell_k)$ [Kociumaka et al. 2024, Cor. 3.9], thus the number of levels k is $O(\log m)$ and the total size of the parse is $O(m)$. Kociumaka et al. [2024, Sec. 5.2], modifying the method described by Christiansen et al. [2020, Sec. 6.2], show how to obtain all the sequences P_k in $O(m)$

time: they start from[3] $P_0 = P$ and, in every level, use the runs and local minima rules to define the blocks (Section 5). They use perfect hash functions to find the nonterminals assigned to those blocks when they appeared in T. Those hash functions are built at indexing time.

As the window $P[i\,..\,j]$ slides over P, we will maintain indices i_k and j_k for all k, so that $P_k[i_k\,..\,j_k]$ is the minimal window of P_k covering $P[i\,..\,j]$ (for $k = 0$ this is $i_0 = i$ and $j_0 = j$). When j or i increase, we check whether each j_k and i_k must be increased as well (for which we maintain $|exp(A)|$ for every nonterminal A).

While the total time to maintain the parse is clearly in $O(m \log m)$, it can also be shown to be $O(m)$. The reason is that, when j (resp., i) increases, we verify from $k = 0$ upwards if j_k (resp., i_k) needs to increase in order to minimally cover $P[i\,..\,j]$. Because all the cuts in P_{k+1} are also in P_k, we can stop as soon as j_k (resp., i_k) does not increase, without need to verify the higher values of k. Therefore, each unit of work in P_{k+1} can be charged to a new reached block $P_k[j_k]$ (resp., to a just abandoned block $P_k[i_k]$). All the work then adds up to $O(m)$, the total length of all the strings P_k.

Note that, even if P appears in T, the parsing of P may generate blocks that do not occur in T, and thus we will not have nonterminals for those in the grammar of T. We assign arbitrary unused permutation values π_k (e.g., larger than all the existing ones) to those new blocks. Blocks in higher levels containing those new symbols will be necessarily new as well. As explained in Section 5, the generated grammar is still of size $O(m)$ and height $O(\log m)$. Further, the search algorithm stays correct, because the new blocks we find and the symbols we assign them form a valid parse of the text $T' = T\$^n P$, where $\$$ does not appear elsewhere in T or P: in S_1 we would have formed the run $\$^n$, whose length would have paused it until the end of the process, so P would be parsed separately of T. If our parse of P missed an occurrence in T, then it would also miss an occurrence in T', where all the blocks found in the parsing of the pattern are known.

We need dynamic data structures to maintain those new symbols we produce, so as to recognize them if they reappear in P_k. We must store keys (a, ℓ) associated with runs a^ℓ of odd levels, and tries for the blocks of the even levels. Since both the values (a, ℓ) and the number of children in trie nodes belong to discrete universes of size $O(m)$, we can use dynamic predecessor data structures [Pătraşcu and Thorup 2006] to operate in time $O(\log \log m)$ in all cases. Added over the $O(m)$ symbols we process along the parse, the time to create and search unknown blocks is bounded by $O(m \log \log m)$.

6.2. Maintaining the cutting points and R

We compute $M(P[i\,..\,j])$ as follows. Per its definition, $M_k(i, j)$ contains the positions in P of blocks ending within $P_k[i_k\,..\,j_k - 1]$ that (i) belong to $[i_0\,..\,i_0 + 2\alpha_{k+1} - 1]$, (ii) are the leftmost in $[i_0 + 2\alpha_{k+1}\,..\,j - 1]$, or (iii) belong to $[j_0 - \alpha_{k+1}\,..\,j_0 - 1]$. Those are collected consecutively to the right of $P_k[i_k]$ and to the left of $P_k[j_k]$. Further, there are a constant number of such block endings in every level k, because a segment of length $O(\alpha_{k+1})$ contains only $O(1)$ block endings in P_k [Kociumaka et al. 2024, Lem. 3.8].

We merge all the sets $M_k(i, j)$ to form $M(i, j) = M(P[i\,..\,j])$. This set is easily created in sorted order and removing duplicates, because all the block ends of $P_{k+1}[i_{k+1}\,..\,j_{k+1} - 1]$ also exist in $P_k[i_k\,..\,j_k - 1]$. Therefore for (i) we must collect the block ends of $P_0[i_0\,..\,j_0 - 1]$ covering $[i_0\,..\,i_0 + 2\alpha_1 - 1]$, then append the block ends of $P_1[i_1\,..\,j_1 - 1]$ covering $[i_0 + 2\alpha_1\,..\,i_0 + 2\alpha_2 - 1]$, then the blocks ends of $P_2[i_2\,..\,j_2 - 1]$ covering $[i_0 + 2\alpha_2\,..\,i_0 + 2\alpha_3 - 1]$, and so on. For (ii), we collect the sequence of the leftmost block ends in $[i_0 + 2\alpha_{k+1}\,..]$, which is increasing with k. Finally, for (iii) we proceed

[3]They actually start from $P_0 = \#P\$$, surrounding P with new symbols, but this was inherited from the previous parse [Christiansen et al. 2020] and not really needed in theirs [Kociumaka et al. 2024].

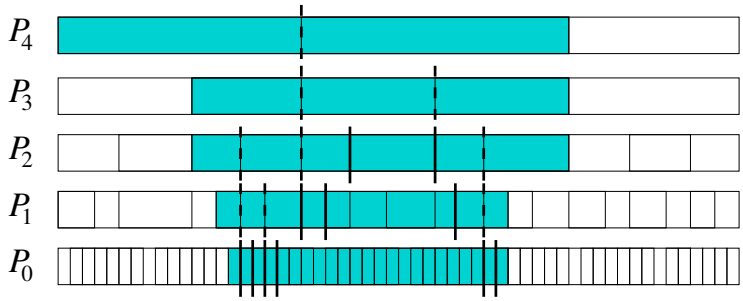

Fig. 7. A schematic view of a parsing of P, shading the involved ranges $P_k[i_k\,..\,j_k]$, and the block ends that make it to $M_k(i,j)$, assuming they happen to be the first 4 and the last 2 in each level. Dashed segments indicate block ends that are already reported in a lower level.

analogously as for (i), prepending for increasing k the block ends of $P_k[i_k\,..\,j_k-1]$ that are in $[j_0-\alpha_{k+1}+1\,..\,j_0-\alpha_k]$. Since the three sequences are obtained in increasing order, they can be merged in time $O(\log m)$. See Fig. 7.

Just as in Section 6.1, the total time of this computation can be made $O(m)$ by working only as long as j_k (resp., i_k) increases, so that the constant amount of work around i_{k+1} and j_{k+1} can be charged to the linear traversal of the window $P_k[i_k\,..\,j_k]$. Precisely, if $i_0+2\alpha_{k+1}-1$ (resp., $j_0-\alpha_{k+1}-1$) does not coincide with a block end in P_k, then no further changes will occur in $M_{k'}(i,j)$ for any $k'\geq k$. Further, the changes that occur in the lists $M_{k'}(P[i\,..\,j])$ for $k'<k$ are placed within the $O(k)$ first and last positions in $M(i,j)$, so these updates can be done in $O(1)$ amortized time per increase of j or i. The number of times the loop of lines 13–18 executes, consequently, is also $O(m)$ along the whole process.

6.3. Patricia tree searches

As explained, we will find beforehand the deepest nodes $vx_r\in T_{\mathcal{X}}$ for all $P[..r]^{rev}$ and $vy_r\in T_{\mathcal{Y}}$ for all $P[r+1\,..]$. We make use of the following result, which was key to obtain subquadratic times in grammar-based indexing.

LEMMA 6.1 (CHRISTIANSEN ET AL. [2020, LEM. 6.5]). *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 size $O(|\mathcal{S}|)$ such that, later, given a pattern $P[1\,..\,m]$ and τ suffixes $Q_1,\ldots,Q_\tau$ of P, we find the ranges of strings in (the lexicographically-sorted) set $\mathcal{S}$ prefixed by each Q_i, in $O(m+\tau(f_h(m)+\log m)+f_e(m))$ total time.*

The Karp-Rabin function κ [Karp and Rabin 1987] used in the lemma guarantees no collisions between substrings of T, so the resulting searches are always correct for the suffixes Q_i that occur as a prefix in $\mathcal{S}$; for the others the structure correctly answers Q_i does not occur.

When $\mathcal{S}=\mathcal{X}$ or $\mathcal{S}=\mathcal{Y}$, our data structure $\mathcal{E}$ provides the required prefix/suffix extraction in time $f_e(\ell)=O(\ell)$. We consider next how to compute the signatures κ. We focus on the more complicated case of $Y\in\mathcal{Y}$; the case of $X\in\mathcal{X}$ is analogous.

6.3.1. Computing signatures. A result of independent interest is that we can obtain $f_h(\ell)=O(\log\ell)$ time on our grammar. Recall that, on Karp-Rabin signatures, we can compute in $O(1)$ time one of $\kappa(S\cdot S')$, $\kappa(S)$, and $\kappa(S')$, given the other two [Christiansen et al. 2020, Sec. A.3].

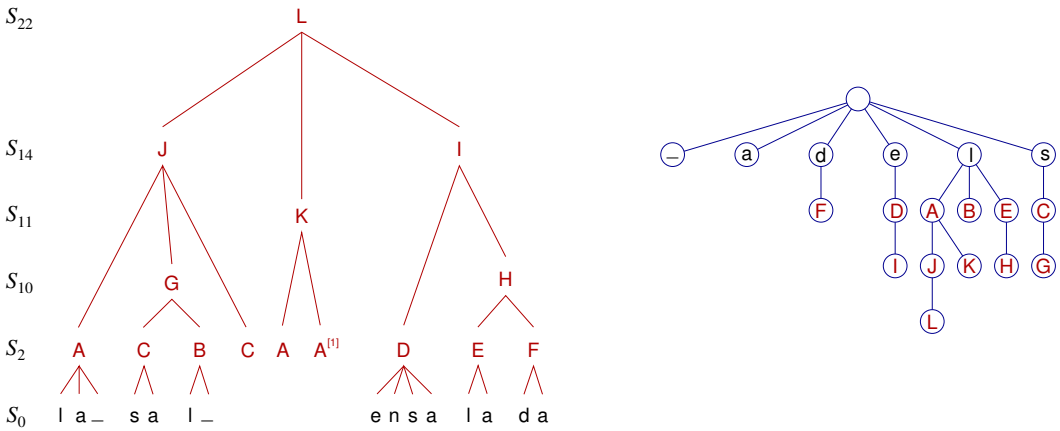

Fig. 8. On the left, the grammar tree of the LCG of Fig. 7, distinguishing the levels where the nodes are formed. On the right, the data structure $\mathcal{E}$ for prefix access. To compute $\kappa(exp(L)[..6]) = \kappa(\text{“la\_sal”})$, we search the ancestors of L in $\mathcal{E}$ for the lowest one with expansion length at least 6, which is J. From J we go towards the 6th leaf, doing the downward path J $\to$ G $\to$ B $\to$ l in the grammar tree, composing the signatures of the left siblings in the path: $\kappa(exp(\mathsf{A}))$, $\kappa(exp(\mathsf{C}))$, and $\kappa(exp(\mathsf{l}))$.

LEMMA 6.2. *The Karp-Rabin signature $\kappa(Y[..\ell])$ of any $Y \in \mathcal{Y}$ can be computed in time $O(\log \ell)$ with our grammar.*

PROOF. We build on the same structure $\mathcal{E}$ used for extraction from the root of $T_{\mathcal{Y}}$. The strings in $\mathcal{Y}$ are concatenations $Y = exp(B_s) \cdots exp(B_t)$ of siblings in rules $A \to B_1 \cdots B_t$ of the grammar tree. The node $v \in T_{\mathcal{Y}}$ of Y stores $\langle v \rangle = B_s$. Let us first assume that $|exp(B_s)| \geq \ell$, so the signature can be computed on $exp(B_s)[..\ell]$.

Structure $\mathcal{E}$ [Gasieniec et al. 2005] is a set of tries on the grammar symbols. The terminals Σ form the trie roots. If $A \to B_1 \cdots B_t$, then B_1 is the parent of A. If $A \to B^t$, then B is the parent of A. Any ancestors C, D of B_s in the tries are nodes that descend from B_s by the leftmost path in the parse tree. The structure $\mathcal{E}$ can jump from B_s to any trie ancestor in constant time. Our grammar is locally balanced: there can be only one block ending inside $exp(B_s[..\ell])$ at levels $k \geq 1 + 2\log_{4/3}(4\ell)$ [Kociumaka et al. 2024, Lem. 3.8]. Thus, the parent node C of the nonterminal D formed at level k has height $d \leq k+1$ in the parse tree and its expansion $exp(C)$ contains $exp(B_s[..\ell])$ as a prefix. The node C can then be found in $O(\log\log \ell)$ time with exponential search on the ancestors of B_s in the trie. Therefore, we have that $exp(B_s)[..\ell] = exp(C)[..\ell]$ and can compute the signature on C instead. See Fig. 8.

The basic algorithm to compute signatures takes time $O(\log^2 \ell)$ [Christiansen et al. 2020, Lem 6.7]. It moves from C towards the leaf L of the parse tree that corresponds to $exp(C)[\ell]$. Let $C \to C_1 \cdots C_t$, then it stores every $w_i(C) = |exp(C_1 \cdots C_i)|$ and every $\kappa_i(C) = \kappa(exp(C_1 \cdots C_i))$. The algorithm finds, in $O(\log i) \subseteq O(\log \ell)$ time, using exponential search, the C_i that is in the path to L (i.e., $w_{i-1} < \ell \leq w_i$), sets $\ell \leftarrow \ell - w_{i-1}$, collects $\kappa_{i-1}(C)$, and continues by C_i. It composes all those κ values towards L to obtain $\kappa(Y[..\ell])$. In rules $C \to C_1^t$ it obtains i in constant time but spends $O(\log i)$ time to compute $\kappa_{i-1}(C)$ from the stored $\kappa(exp(C_1))$.

Instead, an $O(\log n)$ time algorithm [Christiansen et al. 2020, Thm. A.3][4] replaces the exponential searches by a more sophisticated scheme, whose cost is the telescoping sum $\sum_{h=1}^{p} \log(t_h/t_{h-1}) \leq \log t_p$, where t_h is the number of children (counting $C \to C_1^t$

[4]They build on the scheme of Bille et al. [2017] and mention in passing that they could compute the fingerprint in $O(\log \ell)$ time with their grammar, but they had overlooked several issues we fix only now.

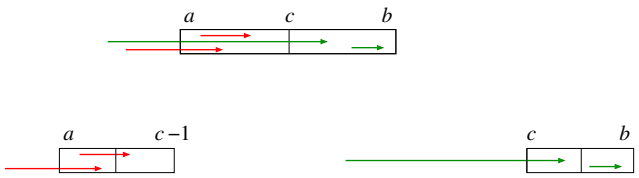

Fig. 9. Illustration of the suffixes that belong to $\mathcal{Q}_{a,b}$ (top). The algorithm only knows that the prefix they share with $\mathcal{Y}$ ends at or after a but does not exceed b. After testing whether they reach c, the suffixes are moved either to $\mathcal{Q}_{a,c-1}$ (bottom left) or to $\mathcal{Q}_{c,b}$ (bottom right), and tested again.

as having t children) of the ancestor at distance h of leaf L. In their case, they start from the root, which could have $t_p = n$, but if we start it from a node C, its time is $\log t_p \leq \log |exp(C)|$. Another component of the cost is the number of times one leaves from heavy paths; this is again $O(\log n)$ in general but just $O(d) = O(\log \ell)$ if we start from the position of C in its heavy path.

It could be, however, that $exp(C)$ is as long as n. Because it was formed in S_k, however, the child $D = C_i$ of C belongs to $\mathcal{A}_k$ (only those nonterminals are allowed to form rules in S_k), and thus by definition $|exp(C_i)| \leq \ell_k$ and $\log |exp(C_i)| = O(k) = O(\log \ell)$. We can then find i and compute $\kappa_{i-1}(C)$ in time $O(\log i) \subseteq O(\log \ell)$ with the basic method [Christiansen et al. 2020, Lem 6.7] and then continue from C_i, where the more sophisticated technique [Christiansen et al. 2020, Thm. A.3] completes the computation in another $O(\log |exp(C_i)|) \subseteq O(\log \ell)$ time.

In case $|exp(B_s)| < \ell$, we find the first $s < i \leq t$ such that $w_i(A) \geq \ell$, and compute instead the signature of $exp(B_i)[\,.\,\ell - w_{i-1}(A)]$, to then compose it with the stored values $\kappa_{s-1}(A)$ and $\kappa_{i-1}(A)$ to obtain the final signature $\kappa(Y[\,.\,\ell]) = \kappa(exp(A)[w_{s-1}(A) + 1\,..\,w_{s-1}(A) + \ell])$. □

6.3.2. Batched searches. The search for all the nodes vy_r, $1 \leq r < m$, corresponds to searching $T_{\mathcal{Y}}$ for every suffix $P[r+1\,..]$ and returning the deepest reached node. Note that Lemma 6.1 does not yield the node vy_r, but rather its corresponding range $[vy_r^1, vy_r^2]$. By performing a lowest common ancestor (LCA) query on $T_{\mathcal{Y}}$ from the vy_r^1th and vy_r^2th leaves, we obtain $v = lca(vy_r^1, vy_r^2)$ (identifying leaves with their ranks). The answer is indeed $vy_r = v$ if $|v| = m - r$; if $m - r < |v|$ then vy_r is the virtual node of string length $m-r$ on the edge of $T_{\mathcal{Y}}$ that leads to v. Linear-space LCA data structures built in linear time and answering lca in $O(1)$ time are well known [Bender et al. 2005].

It seems Lemma 6.1 will perform the desired searches in time $O(m \log m)$. The problem is that the lemma works only if $P[r+1\,..]$ actually prefixes some string in $\mathcal{Y}$. Otherwise, unlike classical trie searching, it does not yield the maximum prefix of $P[r+1\,..]$ that prefixes some string in $\mathcal{Y}$. We will resort to, essentially, binary searching for those longest prefixes using Lemma 6.1 as an internal tool.

Assume m is a power of 2 for simplicity; the general case is easily deduced. We define sets $\mathcal{Q}_{a,b}$ of positions, containing those values r such that $P[r+1\,..\,a]$ is known to prefix a string in $\mathcal{Y}$ and $P[r+1\,..\,b+1]$ is known not to prefix a string in $\mathcal{Y}$ (if $r+1 > a$, then $P[r+1\,..\,a] = \varepsilon$ prefixes all strings in $\mathcal{Y}$). We start with the set $\mathcal{Q}_{1,m} = \{1, \ldots, m\}$. To process a set $\mathcal{Q}_{a,b}$, we search for all the $\tau = |\mathcal{Q}_{a,b}|$ suffixes $\{P[r+1\,..\,c],\ r \in \mathcal{Q}_{a,b}\}$ of $P[\,.\,.\,c]$ using Lemma 6.1, with $c = (a+b+1)/2$ (which is always integer because $b-a+1$ is always a power of 2). The values r where $P[r+1\,..\,c]$ is found are moved to $\mathcal{Q}_{c,b}$, and the others to $\mathcal{Q}_{a,c-1}$ (if $r+1 > c$, we directly move r to $\mathcal{Q}_{c,b}$ without searching for it). Whenever $P[r+1\,..\,c]$ is found in $\mathcal{Y}$, we will associate with r the corresponding node $v_{r,c} \in T_{\mathcal{Y}}$; when $P[r+1\,..\,c]$ is not found, we will retain their previous node $v_{r,a}$ instead. (In the beginning all such nodes are $v_{r,r}$ and equal the root of $T_{\mathcal{Y}}$.) See Fig. 9

Note that the values $b-a+1$ halve as the elements in $\mathcal{Q}_{a,b}$ are separated into two sets. Any value r is then moved $O(\log m)$ times until it ends up in a set of the form $\mathcal{Q}_{c,c}$; at this point we know that the longest prefix of $P[r+1\,..]$ that is also a prefix in $\mathcal{Y}$ is $P[r+1\,..\,c]$, and also know its node $v_{r,c} = vy_r$.

The cost of using Lemma 6.1 has two parts. The cost $f_h(m) + \log m = O(\log m)$ can be charged to each of the τ suffixes sought, and there is an additional global cost of $m + f_e(m) = O(m)$. Since every suffix $P[r+1\,..]$ participates $O(\log m)$ times in the lemma, the first cost adds up to $O(m \log^2 m)$ over all the $m-1$ positions r. The second part is potentially very large, however: the suffixes in $\mathcal{Q}_{a,b}$ may start well to the left of a, thus the pattern is $P[..\,c]$, not $P[a\,..\,c]$; a simple application of the lemma would lead to a quadratic cost again. We address this problem next.

6.3.3. Smarter substring extractions. To reduce this time, we consider where the $O(m)$ cost in Lemma 6.1 comes from. A first part refers to the time needed to compute the Karp-Rabin signatures for all the suffixes in $\mathcal{Q}_{a,b}$. This cost is easily maintained within $O(m)$ overall because we can compute the signatures $\kappa(P[r+1\,..])$, for all $1 \le r < m$, in a single pass over P, and then any $\kappa(P[r+1\,..\,j])$ is obtained in constant time from $\kappa(P[r+1\,..])$ and $\kappa(P[j+1\,..])$.

The second part of the $O(m)$ cost corresponds to the time $f_e(m)$ to verify the longest suffix among those that passed some previous filters; the rest of the verification is built on that extracted suffix. Let $P[r+1\,..\,c]$ be the longest candidate suffix. If $r+1 > a$, we extract the actual suffix $P[r+1\,..\,c]$ regularly in time $f_e(c-r) = O(b-a)$ with $\mathcal{E}$, because $P[r+1\,..]$ starts at the root of $T_{\mathcal{Y}}$.

Otherwise, $r+1 \le a$ and thus $P[r+1\,..\,a]$ had been successfully matched before and we have its node $v_{r,a} \in T_{\mathcal{Y}}$. As mentioned, the process of Lemma 6.1 performs several checks before doing the final extraction of the longest suffix surviving the checks. We will add a new check to those, which can only speed up the process: the candidate node v for $P[r+1\,..\,c]$ must now descend from $v_{r,a}$ in order to be further considered. The descendance check is performed in constant time by comparing the leaf range $[v^1, v^2]$ of v with that of $v_{r,a}$. If v passes the test, we know that it does start with $P[r+1\,..\,a]$, and then only need to extract $P[a+1\,..\,c]$ from the text, which is of length $O(b-a)$.

This time, however, the string to extract does not start at the root of $T_{\mathcal{Y}}$, and thus it requires a random access to T.[5] Say that the node v to verify represents the string $Y = exp(B_s)\cdots exp(B_t) \in \mathcal{Y}$, of which we want to extract $Y[a-r+1\,..\,c-r]$ to compare it with $P[a+1\,..\,c]$. Recall, as in Lemma 6.2, that the strings in $\mathcal{Y}$ are concatenations $Y = exp(B_s)\cdots exp(B_t)$ of consecutive siblings in rules $A \to B_1 \cdots B_t$ of the grammar tree (if $A \to B^t$, then the node stores $\langle v\rangle = B^{[t-1]}$ and we have $B_s = B$). Let us first assume that $|exp(B_s)| \ge c-r$, so the substring to extract is within $exp(B_s)[..\,c-r]$. We use again the structure $\mathcal{E}$, now to extract the string in time $O(b-a+\log c)$.

As in Lemma 6.2, we can search in $O(\log\log c)$ time for the lowest descendant C of B_s such that $|exp(C)| \ge c-r$; its height is $d = O(\log c)$ because the grammar is locally balanced. Since $exp(B_s)[..\,c-r] = exp(C)[..\,c-r]$, we descend from C to the leaf L in the parse tree representing $exp(C)[a-r+1]$. Using the same techniques as in Lemma 6.2, the time is $O(d) = O(\log c)$. From L, $exp(C)[a-r+1\,..\,c-r] = Y[a-r+1\,..\,c-r]$ is extracted in time $O(c-a) = O(b-a)$.

[5] It is tempting to say that, since we had already matched $P[r+1\,..\,a]$ from the root of $T_{\mathcal{Y}}$, we could somehow save the state of that extraction so as to continue without paying the overhead of the random access. However, we might have never extracted the text of the node $v_{r,a}$ explicitly; its verification may have been carried out as a subproduct of reading a longer suffix, starting before $r+1$.

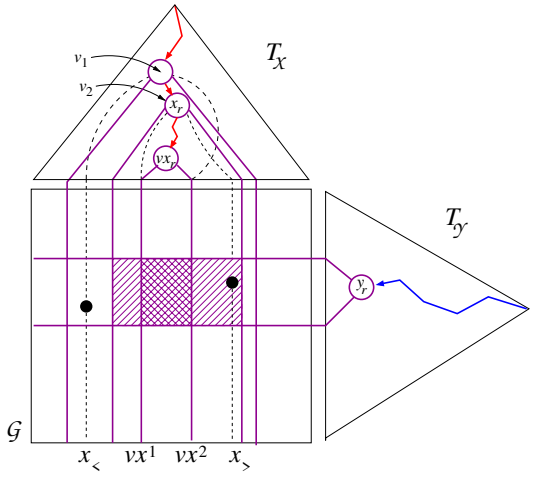

Fig. 10. Computing the lowest ancestor x_r of vx_r in $T_{\mathcal{X}}$ for a given node $y_r \in T_{\mathcal{Y}}$. The initial empty area of vx_r is doubly hatched, and it is extended to the hatched nonempty area of x_r.

In case $|exp(B_s)| < c - r$, the node C is not a descendant of B_s but we use $C = A$ instead. Given the limitation on $|exp(B_s)|$, the height of B_s is $O(\log c)$, and so is the height of A.

The $O(\log c) \subseteq O(\log m)$ cost is absorbed by other $O(\log m)$ costs we have already considered. To account for the $O(b - a)$ terms, let us arrange the sets $\mathcal{Q}_{a,b}$ into *levels*; all the sets with the same difference $b - a + 1$ belong to the same level. Concretely, level 0 corresponds to $\mathcal{Q}_{1,m}$, and if $\mathcal{Q}_{a,b}$ is of level ℓ, then its halves $\mathcal{Q}_{a,c-1}$ and $\mathcal{Q}_{c,b}$ are of level $\ell + 1$. Level ℓ is then formed by 2^ℓ sets whose ranges $[a, b]$ partition $[1, m]$. Therefore, the sum of the $b - a$ values over every level yields m, and adding over all the $O(\log m)$ levels we have that the $O(b - a)$ costs add up to $O(m \log m)$.

6.4. Range successor queries

Finally, we face the problem of, given a y-coordinate range $[y_r^1, y_r^2]$ and the x-coordinate range $[vx_r^1, vx_r^2]$ of the deepest node $vx_r \in T_{\mathcal{X}}$ reached by $P[..r]^{rev}$, find the lowest ancestor x_r of vx_r such that the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ in $\mathcal{G}$ is nonempty.

We will solve this query using (orthogonal) *range successor queries*: given a range $[vx_r^1, vx_r^2] \times [y_r^1, y_r^2]$, we can find the largest value $x_< \leq vx_r^1$ such that $[x_<, vx_r^2] \times [y_r^1, y_r^2]$ contains a point, and the smallest value $x_> \geq vx_r^2$ such that $[vx_r^1, x_>] \times [y_r^1, y_r^2]$ contains a point. Those queries run in $O(\log^\epsilon g)$ time on a grid with g points, using an $O(g)$-space data structure, for any constant $\epsilon > 0$ defined at construction [Nekrich and Navarro 2012]; its construction time is $O(g\sqrt{\log g})$ [Belazzougui and Puglisi 2016].

The lowest ancestor x_r of vx_r containing some point in $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ must then satisfy $x_r^1 \leq x_<$ or $x_r^2 \geq x_>$. In the first case, it is $v_1 = lca(x_<, vx_r^2)$; in the second, it is $v_2 = lca(vx_r^1, x_>)$. Both v_1 and v_2 are ancestors of vx_r, and thus of each other. The correct node x_r is then the lowest of v_1 and v_2, which is known from the leaf ranges stored at the nodes. Fig. 10 illustrates the process.

With this query, line 13 of Algorithm 2 does not cycle; it just performs one $O(\log^\epsilon g)$-time step. It can then be counted as one of the $O(|R|)$ operations performed in each cycle j. Since there are $O(m \log m)$ such operations, this one adds $O(m \log m \log^\epsilon g)$ to the total cost.

```
while l ≤ j + 1 do
    q ← 0 ;
    for r ∈ R do
        if l = r − |x_r| + 1 then q ← q + sum of occurrences in [x_r^1, x_r^2] × [y_r^1, y_r^2] ;
    end
    if q ≥ k then break ;
    for r ∈ R do
        if l = r − |x_r| + 1 then
            x_r ← (virtual) parent of x_r ;
            if |x_r| = 0 then R ← R \ {r} ;
        end
    end
    l ← l + 1
end
```

Algorithm 4: Modification to find the k-MEMs of $P[1\,..\,m]$ in T using a grammar-based index.

6.5. The final result

As stated, our time complexities add up to $O(m(\log^2 m + \log\log n + \log m \log^\epsilon g_{rl}))$ for a RLCFG of size g_{rl}. Since in our case $g_{rl} = O(\delta \log \frac{n \log \sigma}{\delta \log n})$, we can write the time as $O(m \log m(\log m + \log^\epsilon \delta + \log\log n))$. The construction time of all the data structures we use is dominated by the $O(n \log n)$ expected time needed to build the Karp-Rabin hashes of Lemma 6.1 [Christiansen et al. 2020, Sec. 6.6] (the grammar is built in $O(n)$ expected time, see Kociumaka et al. [2024, Cor. 3.16]).

THEOREM 6.3. *Let $T[1\,..\,n]$, over an alphabet of size σ, have repetitiveness measure $\delta = \delta(T)$. Then, for any constant $\epsilon > 0$, we can build a data structure of size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ that finds the MEMs of any given pattern $P[1\,..\,m]$ in time $O(m \log m(\log m + \log^\epsilon \delta + \log\log n)) \subseteq O(m \log m(\log m + \log^\epsilon n))$, returning an occurrence of each MEM. The data structure can be built in $O(n \log n)$ expected time. The query process uses $O(m)$ additional space on top of the data structure.*

We have assumed $m \leq n$ throughout. If not, we can just extract T from the grammar, build a suffix tree on it, and run the classic algorithm, all in $O(n+m) = O(m)$ expected time (the expectation owes to the suffix tree construction), or $O(m \log \min(m, \sigma))$ worst-case time if we do not build perfect hash functions on the children of suffix tree nodes.

7. RELATED PROBLEMS

We now apply our findings to solve some problems related to finding MEMs. In all cases we will still consider that T is indexed with a grammar, whereas P is given in online form and we can spend $O(m)$ space at query time.

7.1. Finding k-MEMs

We first extend our results to finding the k-MEMs of $P[1\,..\,m]$, where k is given at query time together with P. The idea is to count the number of secondary occurrences triggered from the primary occurrences we find in the grid. The total number of occurrences of $P[i\,..\,j]$ in T is the sum of those numbers over all the points in the areas $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ corresponding to the active positions $r \in R$.

Algorithm 4 shows the lines we must insert right after line 18 of Algorithm 2 to account for k-MEMs. In addition, in line 24 of Algorithm 2 we must collect some of the points reached in line 4 of Algorithm 4, instead of one of $r_{\min}$.

The new invariants for the window $P[i\,..\,j]$ are that (i) we have reported every k-MEM ending before j, and that either (ii.1) $R = \emptyset$ and no k-MEM of $P[..\,j]$ ends at j, or (ii.2) $R \neq \emptyset$ and the k-MEM of $P[..\,j]$ ending at j is (i, j).

The required changes, in Algorithm 4, are that once we compute the leftmost position l such that $P[l\,..\,j+1]$ occurs in T, we check in lines 2–5 that there are at least k occurrences of $P[l\,..\,j+1]$. Only the occurrences of the active positions r that reach position l (i.e., $l = r - |x_r| + 1$) are counted. If we count less than k occurrences, then we must advance l, going to the parent of all those nodes x_r that reached l, and removing the active positions r where x_r becomes the root of $T_{\mathcal{X}}$.

The dominant term in the time complexity are the queries that count the sum of occurrences in orthogonal subgrids $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$. This query is performed $|R|$ times for every new value of j and of i (or l, in our case). With the bound $|R| \leq m$, we have a total time of $O(\tau m^2)$, where τ is the time of the geometric query.

The counting query can be done by performing up to k orthogonal range successor queries on $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$, finding one primary and zero or more secondary occurrences from each point we find in the grid. Note that, although the process to find the secondary occurrences ensures only amortized time [Claude et al. 2021], we need to know only the number of such occurrences, so we know we have k occurrences as soon as we perform k recursive calls along the process of finding the secondary occurrences. In general, we stop as soon as we add up to k occurrences along all the primary and secondary occurrences triggered by all the points found in the consecutive range successor queries on $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$. Since each point brings in at least one primary occurrence, and the range successor queries take $O(\log^\epsilon g_{rl})$ time, we obtain $\tau = O(k \log^\epsilon g_{rl})$. This yields a natural generalization of of Theorem 4.2.

THEOREM 7.1. *The data structure of Theorem 4.2 can find the k-MEMs of any given pattern $P[1\,..\,m]$, for any $k > 0$ given with P, in time $O(km^2 \log^\epsilon g_{rl})$.*

Algorithm 4 can also be inserted right after line 24 of Algorithm 3. Now the size of R is bounded by $O(\log m)$, which makes Algorithm 4 run in time $O(\tau m \log m)$.

THEOREM 7.2. *The data structure of Theorem 6.3 can find the k-MEMs of any given pattern $P[1\,..\,m]$, for any $k > 0$ given with P, in time $O(m \log m(\log m + k(\log^\epsilon \delta + \log\log n))) \subseteq O(m \log m(\log m + k \log^\epsilon n))$.*

These solutions, however, are impractical for large k. We can improve the time of Theorem 7.1 by restricting our grammar to be a CFG, of size $g \geq g_{rl}$. On CFGs, the number of secondary occurrences triggered by each primary occurrence is a function of the grid point only, and thus it can be precomputed and stored with the points. The summing query can then be solved directly as a geometric query, in time $O(\log^{2+\epsilon} g)$ for any constant $\epsilon > 0$ chosen at index construction time, within $O(g)$ space [Navarro 2019, Sec. 6]. The construction time does not change with respect to the basic solution. This is then faster than the solution of Theorem 7.1 when $k = \omega(\log^2 g)$.

THEOREM 7.3. *Assume we have a CFG of size g that generates only $T[1\,..\,n]$. Then, for any constant $\epsilon > 0$, we can build a data structure of size $O(g)$ that finds the k-MEMs of any given pattern $P[1\,..\,m]$, for any $k > 0$ given with P, in time $O(m^2 \log^{2+\epsilon} g)$. The data structure can be built in $O(g \log^2 n)$ time and $O(g \log n)$ space, or in $O(n + g\sqrt{\log g})$ time and $O(n)$ space. The query process uses $O(m)$ additional space on top of the data structure.*

The time can drop to $O(m^2 \log g)$ if we use the $O(g \log g)$ space of a faster and larger geometric representation for g points that answers range counting queries in time $O(\log g)$ [Alstrup et al. 2000]. The structure is built in $O(g \log^2 g)$ expected time.

Although it is unknown how to set the point values in the grid so that they correctly sum up the secondary occurrences on a general RLCFG, the problem can be handled in our particular grammar [Kociumaka et al. 2024, Sec. 6] (see also the technique they build on [Christiansen et al. 2020, Sec. 7], which has more details). The only obstacle to use the technique is that it needs to compute the shortest period $p = p(i,j)$ of any window $P[i\,..\,j]$ we consider (this is the smallest positive value such that $P[i\,..\,j-p] = P[i+p\,..\,j]$). Kociumaka [2018, Thm. 1.1.2] shows how to preprocess P in $O(m)$ time so that any $p(i,j)$ can be computed in $O(\log m)$ time. The total time per counting query is then $\tau = O(\log m + \log^{2+\epsilon} g)$ for a grammar of size g, and the query time is $O(\tau m \log m)$.

THEOREM 7.4. *Let $T[1\,..\,n]$ have alphabet size σ and repetitiveness measure $\delta = \delta(T)$. Then, for any constant $\epsilon > 0$, we can build a data structure of size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ that finds the k-MEMs of any given pattern $P[1\,..\,m]$, for any $k > 0$ given with P, in time $O(m \log m (\log m + \log^{2+\epsilon}(\delta \log \frac{n \log \sigma}{\delta \log n}))) \subseteq O(m \log m \log^{2+\epsilon} n)$, returning an occurrence of each MEM. The data structure can be built in $O(n \log n)$ expected time. The query process uses $O(m)$ additional space on top of the data structure.*

Again, by spending $O(\delta \log \frac{n \log \sigma}{\delta \log n} \log n)$ space, the range counting time drops to $O(\log(\delta \log \frac{n \log \sigma}{\delta \log n}))$ [Kociumaka et al. 2024; Alstrup et al. 2000], and thus the total query time becomes $O(m \log m (\log m + \log(\delta \log \frac{n \log \sigma}{\delta \log n}))) \subseteq O(m \log m \log n)$.

7.2. MUMs

As described in Section 2, a MUM between P and T is a maximal substring that appears exactly once in both P and T. We build on the following observation.

LEMMA 7.5. *If $P[i\,..\,j]$ is a MUM between P and T, then it must be a MEM in T.*

PROOF. If $P[i\,..\,j]$ is a MUM, then it appears exactly once in P and T. If it is not a MEM in T, then either $P[i-1\,..\,j]$ or $P[i\,..\,j+1]$ appear in T as well, and because they cannot be more frequent than $P[i\,..\,j]$, they appear once both in P and in T. Therefore, $P[i\,..\,j]$ is not maximal. □

Our general strategy is then to find the MEMs of P in T and filter out those that are not MUMs. We build the suffix tree of P in $O(m)$ worst-case time [Farach-Colton et al. 2000], without building the perfect hash functions on the children of suffix tree nodes (the lack of perfect hashing will pose an $O(m \log m)$ time overhead to the algorithms, which will be absorbed by higher complexities). Then we run Algorithm 2+4 (i.e., adding to Algorithm 2 the lines of Algorithm 4) for finding k-MEMs with $k = 1$. This seems to be the same as finding MEMs, but we will modify the k-MEM finding algorithm soon.

At the same time, we traverse the suffix tree of P, maintaining the locus of $P[i\,..\,j]$ by using the child and suffix link operations, just as described in Algorithm 1. This time, however, we run on the suffix tree of P itself, not of T. Further, we decide whether to increase j (i.e., go to the child of the current suffix tree node) or to increase i (i.e., go to the suffix link of the current node) following what Algorithm 2+4 does.

We use the implementation of Algorithm 4 used to prove Theorem 7.1. This implementation does not really compute the precise number q of occurrences of $P[i\,..\,j+1]$ in T, but it works just enough to determine if $q < k$ or $q \geq k$ (for us, $q = 0$ or $q = 1$). This allows it to run in time proportional to k. We slightly extend its counting process until

determining whether $q = 0$, $q = 1$, or $q \geq 2$, so that $q = 1$ determines that $P[i\,..\,j+1]$ appears exactly once in T. We then proceed as for k-MEMs with $k = 1$, but in case $q \geq 2$ we record an invalid position in p.

When it comes to report (i, j) and its occurrence $T[p - j + i\,..\,p]$, we do so only if, in addition (a) p is a valid position, which means $P[i\,..\,j]$ appears only once in T, and (b) the locus of $P[i\,..\,j]$ in its suffix tree is (on an edge to) a leaf, which means that $P[i\,..\,j]$ appears only once in P. We apply the same filter when reporting (i, m) in the last line of Algorithm 2.

In the same way we have combined Algorithms 2 and 4, we can combine Algorithms 3 and 4. This yields the following result.

THEOREM 7.6. *The data structures of Theorems 4.2 and 6.3 can find the MUMs of T and any given pattern $P[1\,..\,m]$, in the same asymptotic time they require to find the MEMs. The query process uses $O(m)$ additional space.*

7.3. Rare MEMs

A MEM of P in T is k-rare if it appears in P and T, yet at most k times; MUMs are 1-rare MEMs. We can generalize the method of Section 7.2 so that we still look for k-MEMs with $k = 1$, but work enough on the counting stage so as to determine whether $q = 0$, $1 \leq q \leq k$, or $q > k$. We store an invalid position p when $q > k$; otherwise we proceed as for $k = 1$. When it comes to report (i, j), we do so only if (a) position p is valid (i.e., $P[i\,..\,j]$ appears at most k times in T) and (b) the locus of $P[i\,..\,j]$ in the suffix tree of v is (on the edge to) a node v having at most k descendant leaves (for which we store the number of leaves that descend from the suffix tree nodes, which can be computed in $O(m)$ time once the suffix tree is built).

By counting the occurrences in different ways, we obtain the following result.

THEOREM 7.7. *The data structures of Theorems 7.1, 7.2, 7.3, and 7.4 can find the k-rare MEMs of T and any given pattern $P[1\,..\,m]$, in the same asymptotic time they require to find the k-MEMs of P in T. The query process uses $O(m)$ additional space.*

8. APPLICATIONS

We now describe other popular problems related to finding MEMs our results impact.

8.1. Matching statistics

A problem directly related to finding MEMs is that of computing the so-called matching statistics, which we already discussed in the Introduction. While first defined by Chang and Lawler [1994] for approximate string matching, they have a number of bioinformatic applications like estimating the distance between genomes for phylogenetic reconstruction, estimating cross-entropy, computing string kernels, string mining, and species estimation, to name a few [Ohlebusch 2013; Mäkinen et al. 2015].

Definition 8.1. Given a text $T[1\,..\,n]$ that can be preprocessed, the *matching statistics problem* is that of, given a pattern $P[1\,..\,m]$, return the length $M[q]$ of the longest prefix of $P[q\,..]$ that occurs in T, for every $1 \leq q \leq m$. A position where each such longest prefix occurs must be given for each q.

Given a solution to the MEM finding problem, $(i_1, j_1), \ldots, (i_s, j_s)$ with increasing values i_r (and j_r), we compute the matching statistics as follows. Set all $M[q]$ to zero and then traverse the tuples (i_r, j_r) in order. Set $M[q] = j_r - q + 1$ for all $i_r \leq q \leq \min(j_r, i_{r+1} - 1)$, assuming $i_{s+1} = m + 1$. The occurrence of each $M[q] > 0$ is that of its MEM (i_r, j_r) shifted by $q - i_r$. Conversely, given the matching statistics $M[q]$ for $1 \leq q \leq m$, we obtain the MEMs by reporting, for increasing i, every pair $(i, i + M[i] - 1)$ such that $i = 1$ or $M[i] \geq M[i - 1]$, and $M[i] > 0$.

Therefore, both problems are interchangeable as one output can be converted to the other in optimal $O(m)$ time. In particular, our results allow computing the matching statistics of a patterm $P[1..m]$ with respect to a text $T[1..n]$ that is indexed in compressed form, within the same space and times seen in Theorems 4.2 and 6.3.

8.2. MEMs against a collection of texts

Some applications, especially related to multiple alignment, consider computing the MEMs and k-MEMs between a pattern P and a collection of texts $T_1, \ldots, T_\tau$ [Gusfield 1997, Sec. 7.6], and even more prominently, the MUMs [Mäkinen et al. 2015, Ch. 11].

Definition 8.2. The k-MEMs between P and a collection of texts $T_1, \ldots, T_\tau$ are the maximal substrings of P that appear at least k times in each of the texts T_t, for $1 \leq t \leq \tau$. The case $k = 1$ corresponds to the MEMs between P and $T_1, \ldots, T_\tau$. The k-rare MEMs are the maximal substrings of P that appear between 1 and k times, both in P and in each text T_t, for $1 \leq t \leq \tau$. The case $k = 1$ corresponds to MUMs.

While we only consider finding the MEMs and k-MEMs between a pattern P and a text T, assume that we have the texts T_t in compressed form and we compute the list of (k-)MEMs $M_t = (i^t_1, j^t_1), \ldots, (i^t_{s_t}, j^t_{s_t})$ between P and every T_t. Those lists come sorted by increasing i^t and by increasing j^t. We then compute the (k-)MEMs between P and the collection by running a segment intersection algorithm on the lists M_t: we want the maximal segments that are included in a segment of each M_t.

We set cursors $c_t = 1$ for all t, and compute $i = \max\{i^t_{c_t}, 1 \leq t \leq \tau\}$ and $j = \min\{j^t_{c_t}, 1 \leq t \leq \tau\}$. If $i \leq j$ we report the (k-)MEM (i, j). We then increment the cursor(s) c_t for which $j^t_{c_t} = j$, and continue until exhausting some of the lists. By using a max-heap to maintain the values $i^t_{c_t}$ and a min-heap to maintain the values $j^t_{c_t}$, the whole process takes time $O(N \log \tau)$, where N is the total length of the lists M_t. This complexity is negligible compared to that of computing the (k-)MEMs.

Finding MUMs and k-rare MEMs between P and texts $T_1, \ldots, T_\tau$ requires intervening the combined Algorithm 3+4 we developed in Sections 7.2 and 7.3. Concretely, we run in synchronization the algorithm for every T_t, apart from moving in the suffix tree of P. We maintain the same interval $P[i..j]$ on all the texts, skipping the value of $j + 1$ if the condition in line 5 of Algorithm 3 is true for any of the texts. Further, the value l computed in line 24 must correspond to the maximum l over all the texts. For that value of l, we will determine if any of the texts T_t has more than k occurrences of $P[i..j]$, and if so, make the text position invalid so that this window is not reported. The asymptotic cost is then the same as the sum of the costs of computing MUMs of k-rare MEMs between P and every text T_t.

A weakness of this approach is that the texts T_t must be indexed separately, which does not exploit their joint repetitiveness. If all the texts are indexed together, even the problem of determining if a given pattern appears in all the documents is unexplored for grammar-based indices (whereas it is easy to solve with suffix-based ones [Sadakane 2007], by counting in how many texts the pattern appears).

8.3. Relative Lempel-Ziv compression

Relative Lempel-Ziv (RLZ) [Kuruppu et al. 2010] is a compression algorithm that first chooses a *reference* text R, and then compresses texts T by representing them as minimum-length sequences of substrings of R. That is, RLZ represents T in $O(z)$ space by choosing a minimal number z of substrings of R such that $T = R[a_1..b_1] \cdots R[a_z..b_z]$.[6] RLZ has proved particularly effective to compress collections

[6] We assume R contains every alphabet symbol, so T can always be covered by substrings of R.

where every text is relatively close to each other, such as in genome collections of the same species [Kuruppu et al. 2010] and web pages [Hoobin et al. 2011].

The compression algorithm traverses $T[1\,..\,n]$ left to right, in streaming mode. If it has already compressed $T[1\,..\,i-1]$, it finds the largest $j \in [i\,..\,n]$ such that $T[i\,..\,j]$ occurs in R, say at $R[a\,..\,b] = T[i\,..\,j]$, and outputs the pair (a, b). Finding those longest prefixes $T[i\,..\,j]$ of $T[i\,..\,n]$ in R can be done in time $O(j-i+1)$ with a suffix tree of R, for a total expected compression time of $O(n)$. More space-efficient implementations are possible with compressed suffix arrays, where the matching of $T[i\,..]$ can proceed left-to-right until finding a mismatch. Using a grammar-compressed index for R, instead, is difficult because it is designed to match a fixed $T[i\,..\,j]$ with j known beforehand.

An issue with RLZ is how to choose the reference R. In genome collections of the same species, choosing one random genome is efficient enough, but there are more complex situations, such as metagenomic collections or other versioning scenarios. The general premise is to aim for a relatively small R, especially because we need it indexed in main memory in order to efficiently compress T. Various techniques to choose or generate a good reference have been studied [Kuruppu et al. 2011a; 2011b; Liao et al. 2016; Gagie et al. 2016]. On the other hand, however, the larger R, the better compression we achieve in general, because we can potentially choose longer phrases.

With our techniques, we could represent R in grammar-compressed form, thus R could be a much larger, repetitive, set of texts without compromising the main memory space used by the index of R. For example, we could choose the whole set of genomes in a collection as the reference, so as to compress a new genome T by choosing pieces from the closest possible individuals. Our MEM-finding technique is precisely what we need to simulate the searches for prefixes $T[i\,..]$ with unknown ending positions.

To compress T, we find the MEMs of T in R, $(i_1, j_1), \ldots, (i_s, j_s)$. By assuming as before that R contains every alphabet symbol, it holds that $i_1 = 1$, $i_{r+1} \leq j_r$ for all r, and $j_s = n$. The minimum coverage of T then contains $z \leq s$ substrings, which can be built from the MEMs in $O(s)$ time as follows: set cursors at $c = 1$ and $p = 1$. Output pair (p, j_c). Set $p = j_c + 1$. Increment c as long as $i_c \leq p$. Continue until $p = n + 1$. We can then compress T in time $O(n \log n(\log n + \log^{\epsilon} r))$, where $r = |R|$, using Theorem 6.3; the time to cover T with the MEMs is asymptotically irrelevant.

8.4. All-pairs suffix-prefix matches

When assembling a genome from a set of reads without the help of a reference genome (a.k.a., de novo fragment assembly), one relevant technique is to form a so-called "overlap graph", where every node is a read and there is an edge of weight ℓ from node u to node v iff the ℓ-length suffix of the read of u equals the ℓ-length prefix of the read of v. One then aims, in broad terms, to traverse the overlap graph in a way that maximizes the weights and collects all the reads [Setubal and Meidanis 1997, Ch. 4] [Gusfield 1997, Sec. 7.10]. A similar technique is used for transcript assembly [Mäkinen et al. 2015, Ch. 15]. To build the overlap graph, one needs to find all the suffix-prefix matches between every pair of reads, disregarding those whose lengths are below a significance threshold $\ell_{\min}$. This problem, known as the "all-pairs suffix-prefix matching problem", can be solved in optimal time $O(n + e)$, where n is the total length of the reads and e is the number of edges in the resulting graph, yet it needs to build a suffix tree on the concatenation of the reads [Mäkinen et al. 2015, Sec. 8.4.4] [Gusfield 1997, Sec. 7.10.1], or an enhanced suffix array [Ohlebusch 2013, Sec. 5.6.7]. If one uses compressed suffix arrays, then the time complexity becomes $O(n \log \sigma + e)$ [Ohlebusch 2013, Sec. 7.7.3].

Since collections of reads with high coverage are repetitive, we can use our LCG of Section 5 to index the whole sequence $T[1\,..\,n]$ of reads, using a special symbol \$ as a terminator for each. In order to find all the suffix-prefix matches in the set, we process each read $P[1\,..\,m]$ against T. We run a specialized version of Algorithm 3, only as long

as $i = 1$ (because we are interested in the prefixes of P). Since the whole P occurs in T, however, it is not necessary to check for this condition; we will just run the loop of line 4 over lines 7–22, processing each whole read P.

For each iteration of the loop, where we have processed the window $P[1..j]$ and now consider $j+1$, we first *simulate* that $P[j+1] = \$$, run the lines 7–22, and report *all* the occurrences of $P[..j] \cdot \$$ in T. That is, we collect all the primary occurrences and also follow all the secondary occurrences for all active points r such that $r - |x_r| + 1 = 1$ (i.e., they match P from position 1). Each such occurrence corresponds to the suffix of some read matching the prefix $P[..j]$.[7] This simulation is done only if $j \geq \ell_{\min}$.

After reporting the matches, we *undo* the updates performed in lines 7–22 when simulating that $P[j+1] = \$$, and *redo* lines 7–22 for the actual value of $P[j+1]$.

Undoing is implemented by just maintaining a copy of the previous set R and all the values j_k, x_r, and y_r, re-stating them after the simulation. The time for simulating $P[j+1] = \$$, undoing the simulation, and then running again with the actual value, adds up to $O(\log m \log^\epsilon n)$. Considering the preprocessing of lines 1–2 and adding over all the reads, the total time is in $O(n \log m(\log m + \log^\epsilon n))$, where m is the average length of a read,[8] plus the number of suffix-prefix matches found. The working space on top of the index is just $O(m)$.

9. CONCLUSIONS

We have obtained improved results, including the first subquadratic algorithm, to find MEMs on parsing-based indices, which are the most promising in terms of space for highly repetitive text collections. While suffix-based indices can preprocess $T[1..n]$ to find the MEMs of $P[1..m]$ in T in time $O(m \log\log n)$, their space is $\Omega(r)$, where r (the number of runs in the BWT of T) is not such a strong measure of repetitiveness [Navarro 2021b]. Our first result is a data structure of size $O(g_{rl})$, where g_{rl} is the size of the smallest RLCFG that generates T. This is currently the best possible space for any structure able to access T with relevant time guarantees [Navarro 2021b]. Our structure finds the MEMs in $O(m^2 \log^\epsilon n)$ time for any constant $\epsilon > 0$. This is very similar to the time of previous work [Gao 2022], which could also run in $O(g_{rl})$ space. Within $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space, we obtain the first subquadratic time, $O(m \log m(\log m + \log^\epsilon n))$, on a particular RLCFG that has local consistency properties. This space is optimal for every n, σ, and δ, though g_{rl} is always $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ and can be $o(\delta \log \frac{n \log \sigma}{\delta \log n})$ in some text families [Kociumaka et al. 2023]. The MEM finding algorithm is adapted to find the MUMs between P and T within the same space and time complexities.

We also considered the extended problem of computing the k-MEMs of P, with k given at query time. The above complexities are generalized by multiplying the $\log^\epsilon n$ terms by k, while retaining the space. For larger k, we obtain $O(m^2 \log^{2+\epsilon} n)$ time within space $O(g)$ given a CFG of size g that generates T, and $O(m \log m \log^{2+\epsilon} n)$ time within space $O(\delta \log \frac{n \log \sigma}{\delta \log n})$. We find k-rare MEMs, which generalize MUMs, within the same space and time complexities of k-MEMs.

Our techniques are presented on a particular locally consistent grammar [Kociumaka et al. 2024] that yields the best complexities, but they would work on others too (possibly without the same worst-case time guarantees). We believe they could be successfully implemented on practical constructions of CFGs [Claude et al. 2021] built with RePair [Larsson and Moffat 2000] or of locally consistent grammars based on in-

[7] Since P occurs in T, there will be a spurious match with itself, which is easily filtered out when found.
[8] This is because $\log m$ and $\log^2 m$ are convex functions of m.

duced suffix sorting [Díaz-Domínguez et al. 2021; Nunes et al. 2022]. Further, even lacking theoretical guarantees, the algorithm for arbitrary RLCFGs will probably be competitive if implemented on Lempel-Ziv based indices [Kreft and Navarro 2013; Ferrada et al. 2018], which are considerably smaller than those based on grammars.

As we have shown in various applications, our development opens the door to running, on parsing-based indices, algorithms that were reserved to suffix-based ones. This is because the classic exact pattern matching based on cutting points requires that the pattern to match is completely specified from the beginning, therefore excluding problems where one must match parts of the pattern as far as possible. The holy grail would be to simulate the suffix tree functionality within $O(g)$ space, where g is the size of the best grammar, from some relevant family, that represents the text, for example $g = O(\delta \log \frac{n \log \sigma}{\delta \log n})$ with our locally-consistent grammars. Currently we can do this only in $O(r \log \frac{n}{r})$ space with the r-index [Gagie et al. 2020], which is an $O(\log^2 n)$ factor away from δ-optimal space [Kempa and Kociumaka 2020]. A very important step towards this goal is the recent index [Kempa and Kociumaka 2023] that computes suffix array entries (and their inverses) in $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space and $O(\log^{4+\epsilon} n)$ time. Reducing the degree of this polylog is however essential to achieve results that are also practical.

Acknowledgements

We thank Tatiana Starikovskaya, Tomasz Kociumaka, and Adrián Goga for useful discussions, as well as a reviewer for careful notes.

REFERENCES

ALSTRUP, S., BRODAL, G., AND RAUHE, T. 2000. New data structures for orthogonal range searching. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*. 198–207.

AMIR, A., LANDAU, G. M., LEWENSTEIN, M., AND SOKOL, D. 2007. Dynamic text and static pattern matching. *ACM Transactions on Algorithms 3,* 2, article 19.

BANNAI, H., GAGIE, T., AND I, T. 2020. Refining the r-index. *Theoretical Computer Science 812*, 96–108.

BELAZZOUGUI, D., CUNIAL, F., KÄRKKÄINEN, J., AND MÄKINEN, V. 2013. Versatile succinct representations of the bidirectional Burrows-Wheeler Transform. In *Proc. 21st Annual European Symposium on Algorithms (ESA)*. 133–144.

BELAZZOUGUI, D. AND PUGLISI, S. J. 2016. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2053–2071.

BENDER, M. A., FARACH-COLTON, M., PEMMASANI, G., SKIENA, S., AND SUMAZIN, P. 2005. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms 57,* 2, 75–94.

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., 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.

BOUCHER, C., CVACHO, O., GAGIE, T., MANZINI, J. H. G., NAVARRO, G., AND ROSSI, M. 2021a. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*. 60–72.

BOUCHER, C., GAGIE, T., I, T., KÖPPL, D., LANGMEAD, B., MANZINI, G., NAVARRO, G., PACHECO, A., AND ROSSI, M. 2021b. PHONI: Streamed matching statistics with multi-genome references. In *Proc. 31th Data Compression Conference (DCC)*. 193–202.

BURROWS, M. AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

CÁCERES, M. AND NAVARRO, G. 2022. Faster repetition-aware compressed suffix trees based on block trees. *Information and Computation 285B*, article 104749.

CHAN, T. M., LARSEN, K. G., AND PATRASCU, M. 2011. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*. 1–10.

CHANG, W. AND LAWLER, E. 1994. Sublinear approximate string matching and biological applications. *Algorithmica 12,* 4/5, 327–344.

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.

CHRISTIANSEN, A. R., ETTIENNE, M. B., KOCIUMAKA, T., NAVARRO, G., AND PREZZA, N. 2020. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms 17,* 1, article 8.

CLAUDE, F., NAVARRO, G., AND PACHECO, A. 2021. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences 118*, 53–74.

CROCHEMORE, M. AND RYTTER, W. 2002. *Jewels of Stringology*. World Scientific.

DELCHER, A. L., KASIF, S., FLEISHMANN, R., PETERSON, J., WHITE, O., AND SALZBERG, S. 1999. Alignment of whole genomes. *Nucleic Acids Research 27,* 11, 2369–2376.

DÍAZ-DOMÍNGUEZ, D., NAVARRO, G., AND PACHECO, A. 2021. An LMS-based grammar self-index with local consistency properties. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*. 100–113.

FARACH-COLTON, M., FERRAGINA, P., AND MUTHUKRISHNAN, S. 2000. On the sorting-complexity of suffix tree construction. *Journal of the ACM 47,* 6, 987–1011.

FARRUGGIA, A., GAGIE, T., NAVARRO, G., PUGLISI, S. J., AND SIRÉN, J. 2018. Relative suffix trees. *The Computer Journal 61,* 5, 773–788.

FERRADA, H., KEMPA, D., AND PUGLISI, S. J. 2018. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 1–8.

GAGIE, T., NAVARRO, G., AND PREZZA, N. 2020. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM 67,* 1, article 2.

GAGIE, T., PUGLISI, S. J., AND VALENZUELA, D. 2016. Analyzing Relative Lempel-Ziv reference construction. In *Proc. 23rd International Symposium on String Processing and Information Retrieval*. 160–165.

GAO, Y. 2022. Computing matching statistics on repetitive texts. In *Proc. 32nd Data Compression Conference (DCC)*. 73–82.

GASIENIEC, L., KOLPAKOV, R., POTAPOV, I., AND SANT, P. 2005. Real-time traversal in grammar-based compressed files. In *Proc. 15th Data Compression Conference (DCC)*. 458–458.

GIULIANI, S., ROMANA, G., AND ROSSI, M. 2022. Computing Maximal Unique Matches with the r-Index. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*. 22:1–22:16.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

HOOBIN, C., PUGLISI, S. J., AND ZOBEL, J. 2011. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment 5,* 3, 265–273.

KÄRKKÄINEN, J., SANDERS, P., AND BURKHARDT, S. 2006. Linear work suffix array construction. *Journal of the ACM 53,* 6, 918–936.

KARP, R. M. AND RABIN, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 2*, 249–260.

KEMPA, D. AND KOCIUMAKA, T. 2020. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 1002–1013.

KEMPA, D. AND KOCIUMAKA, T. 2023. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proc. 64th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 1877–1886.

KEMPA, D. AND PREZZA, N. 2018. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*. 827–840.

KIEFFER, J. C. AND YANG, E.-H. 2000. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory 46,* 3, 737–754.

KOCIUMAKA, T. 2018. Efficient data structures for internal queries in texts. Ph.D. thesis, University of Warsaw.

KOCIUMAKA, T., NAVARRO, G., AND OLIVARES, F. 2024. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica 86*, 1031–1056.

KOCIUMAKA, T., NAVARRO, G., AND PREZZA, N. 2023. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory 69,* 4, 2074–2092.

KREFT, S. AND NAVARRO, G. 2013. On compressing and indexing repetitive sequences. *Theoretical Computer Science 483*, 115–133.

KURUPPU, S., PUGLISI, S. J., AND ZOBEL, J. 2010. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. 201–206.

KURUPPU, S., PUGLISI, S. J., AND ZOBEL, J. 2011a. Optimized Relative Lempel-Ziv compression of genomes. In *Proc 34th Australasian Computer Science Conference (ACSC)*. Australian Computer Society, 91–98.

KURUPPU, S., PUGLISI, S. J., AND ZOBEL, J. 2011b. Reference sequence construction for relative compression of genomes. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*. 420–425.

LARSSON, J. AND MOFFAT, A. 2000. Off-line dictionary-based compression. *Proceedings of the IEEE 88,* 11, 1722–1732.

LIAO, K., PETRI, M., MOFFAT, A., AND WIRTH, A. 2016. Effective construction of relative lempel-ziv dictionaries. In *Proc. 25th International Conference on World Wide Web (WWW)*. 807–816.

MÄKINEN, V., BELAZZOUGUI, D., CUNIAL, F., AND TOMESCU, A. I. 2015. *Genome-Scale Algorithm Design*. Cambridge University Press.

MÄKINEN, V., NAVARRO, G., SIRÉN, J., AND VÄLIMÄKI, N. 2010. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology 17,* 3, 281–308.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

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

MORRISON, D. 1968. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15,* 4, 514–534.

NAVARRO, G. 2016. *Compact Data Structures – A practical approach*. Cambridge University Press.

NAVARRO, G. 2019. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science 777*, 58–72.

NAVARRO, G. 2021a. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys 54,* 2, article 29.

NAVARRO, G. 2021b. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys 54,* 2, article 26.

NAVARRO, G. 2023. Computing MEMs on repetitive text collections. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM)*. article 22.

NEKRICH, Y. AND NAVARRO, G. 2012. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*. 271–282.

NUNES, D., LOUZA, F., GOG, S., AYALA-RINCÓN, M., AND NAVARRO, G. 2022. Grammar compression by induced suffix sorting. *ACM Journal of Experimental Algorithmics 27*, article 1.1.

OHLEBUSCH, E. 2013. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag.

OHLEBUSCH, E., GOG, S., AND KÜGEL, A. 2010. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. 347–358.

OHLEBUSCH, E. AND KURTZ, S. 2008. Space efficient computation of rare maximal exact matches between multiple sequences. *Journal of Computational Biology 15,* 4, 357–377.

PĂTRAŞCU, M. AND THORUP, M. 2006. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*. 232–240.

RASKHODNIKOVA, S., RON, D., RUBINFELD, R., AND SMITH, A. D. 2013. Sublinear algorithms for approximating string compressibility. *Algorithmica 65,* 3, 685–709.

ROSSI, M., OLIVA, M., LANGMEAD, B., GAGIE, T., AND BOUCHER, C. 2022. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology 29,* 2, 169–187.

RUSSO, L. M. S., NAVARRO, G., AND OLIVEIRA, A. 2011. Fully-compressed suffix trees. *ACM Transactions on Algorithms 7,* 4, article 53.

SADAKANE, K. 2007. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms 5*, 12–22.

SETUBAL, J. AND MEIDANIS, J. 1997. *Introduction to Computational Molecular Biology*. PWS Publishing Company.

SUNG, W.-K. 2010. *Algorithms in Bioinformatics: A Practical Introduction*. Chapman & Hall/CRC Press.

TATARNIKOV, I., FARAHANI, A. S., KASHGOULI, S., AND GAGIE, T. 2023. MONI can find k-MEMs. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM)*. article 26.

WEINER, P. 1973. Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory (FOCS)*. 1–11.