Computing MEMs on Repetitive Text Collections

Gonzalo Navarro
Center for Biotechnology and Bioengineering (CeBiB)
Department of Computer Science, University of Chile, Chile

Abstract

We consider the problem of computing the Maximal Exact Matches (MEMs) of a given pattern $P[1\,..\,m]$ on a large repetitive text collection $T[1\,..\,n]$, 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 $O(\delta \log \frac{n}{\delta})$, 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 $\Omega(\delta \log \frac{n}{\delta})$ is a tight lower bound on the compressibility of repetitive texts T, so our structure has optimal size in terms of n and δ.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases grammar-based indices, maximal exact matches, locally consistent grammars, substring complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.22

Funding *Gonzalo Navarro*: Funded by Basal Funds FB0001, Mideplan, Chile, and Fondecyt Grant 1-200038, Chile.

1 Introduction and Related Work

Mutations and experimental sequencing errors make exact pattern matching seldom used in Bioinformatic applications, except possibly for very short patterns and some niche applications [19, 37, 28]. A much more interesting problem is that of 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). This is useful, for example, to find long conserved areas of a gene or to best align a read (where m is typically in the hundreds or thousands) on a reference genome (where n can be in the billions), and even to find similarities between two genomes. In this paper we are interested in the case where T is known in advance and can be indexed.

Finding MEMs is a classic problem in stringology and can be solved in optimal $O(m)$ time using a suffix tree of T [41, 31] (see, e.g., the similar problem of computing matching statistics [19, Sec. 7.8]). Suffix trees, even if using linear space, are too large to maintain in main memory for current text collection sizes, however. The suffix tree of a single human genome, for example, with $n \approx 3 \cdot 10^9$, may take 60GB with a decent implementation. This makes suffix trees hard to use directly on current bioinformatic collections. Even if lower-space alternatives can replace suffix trees for most tasks [28, MEMs in Sec. 11.1.3], this space reduction is still insufficient to face current projects for sequencing millions of human genomes (see `https://b1mg-project.eu`).

A fortunate situation is that many of the fastest growing text collections are highly repetitive [33]. For example, collections of genomes of the same species feature a small percentage of differences between any pair of genomes. Several text indices exploiting repetitiveness to reduce space have appeared [34]. Those indices may take orders of magnitude less space than the raw data, and even more orders less space than a suffix tree on the data.

Those compressed indices support exact pattern matching, that is, they can list all the positions where P occurs in T. While useful, this is less than the full suffix tree functionality, and insufficient to efficiently implement the classic $O(m)$-time MEM finding algorithm.

Compressed suffix trees for highly repetitive text collections do exist, but do not compress that much. Gagie et al. [17] 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 [7] of T. It could find the MEMs in time $O(m \log \frac{n}{r})$ if we run the algorithm 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 [21], it is a weak one [33, 21], 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 [39, 15, 8, 5].

Another trend has been to expand the functionality of a more basic compressed text index for repetitive texts so as to support specific operations, MEMs in our case. Bannai et al. [1] show how to compute matching statistics (from where MEMs are easily extracted in $O(m)$ time) by extending the RLBWT-index [29], 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 $s = O(\log n)$ [4]. Various implementations of this idea [38, 6, 40] 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 [34]. This is natural because those emulate variants of suffix trees or arrays [30], which simplifies the problem of simulating the suffix tree traversal of the classic MEM-finding algorithm. Even the naive algorithm of searching 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 [34]. 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 [33]. 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) [12]. This makes the problem of finding MEMs considerably harder. We are only aware of the results of Gao [18], who computes matching statistics in time $O(m^2 \log^\epsilon \gamma + m \log n)$ using $O(\delta \log \frac{n}{\delta})$ space (for any constant $\epsilon > 0$), or $O(m^2 + m \log\gamma \log\log\gamma + m \log n)$ using $O(\delta \log \frac{n}{\delta} + \gamma \log \gamma)$ space. Here $\delta \leq \gamma$ are lower-bounding measures of repetitiveness [22, 11]. The size $O(\delta \log \frac{n}{\delta})$ matches a tight lower bound on the size of compressed representations of T [25], so a structure of this size uses asymptotically optimal space for every n and δ.

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 $g_{rl} = O(\delta \log \frac{n}{\delta})$ [25]. 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 $P[i \mathinner{.\,.} j]$ 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 [33]. The result essentially matches the first one of Gao, which could also run within $O(g_{rl})$ space.

We further show that, on a particular grammar featuring local consistency properties [24], 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 $P[i \mathinner{.\,.} j]$, and using much more sophisticated techniques to amortize the costs. This grammar is of size $O(\delta \log \frac{n}{\delta})$, optimal for every n and δ, and within this space we sharply break the quadratic time of previous solutions.

2 Maximal Exact Matches (MEMs) and How to Find Them

We assume the usual notation on strings $S[1..n]$ and that the reader is familiar with the concepts related to suffix trees [41, 31, 13]. We start by defining MEMs.

▶ **Definition 1.** *A* Maximal Exact Match (MEM) *of a pattern $P[1..m]$ in a string T is a 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.** *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 (or j). A position where each MEM occurs in T must also be returned.*

The MEM finding problem can be solved in $O(m)$ time with a suffix tree. Algorithm 1 shows how, abstracting away some complications of implementing it on the long edges of suffix trees. The next problem is strongly related to the MEM finding problem.

▶ **Definition 3.** *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 $M[k]$ of the longest prefix of $P[k..]$ that occurs in T, for every $1 \leq k \leq m$. A position where each such longest prefix occurs must be given for each k.*

Given a solution to the MEM finding problem, $(i_1, j_1), \ldots, (i_s, j_s)$, we compute the matching statistics as follows. Set all $M[k]$ to zero and then traverse the tuples (i_r, j_r) in order. Set $M[k] = j_r - k + 1$ for all $i_r \leq k \leq \min(j_r, i_{r+1} - 1)$, assuming $i_{s+1} = m + 1$. The occurrence of each $M[k] > 0$ is that of its (i_r, j_r) shifted by $k - i_r$. Conversely, given the matching statistics $M[k]$ for $1 \leq k \leq m$, 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 can convert one output to the other in optimal $O(m)$ time. Gusfield [19, Sec. 7.8] shows how to compute matching statistics with the suffix tree.

```
i ← 1; j ← 0;
v ← suffix tree root;
while $j < m$ do
    if $v$ has no child labeled $P[j+1]$ then
        $i ← i + 1$; $j ← j + 1$;
    end
    else
        while $j < m$ and $v$ has a child labeled $P[j+1]$ do
            $j ← j + 1$; $v ←$ the child of $v$ by $P[j+1]$;
        end
        report $(i, j)$ with some occurrence of $v$;
        while $i \leq j < m$ and $v$ has no child labeled $P[j+1]$ do
            $i ← i + 1$; $v ←$ the suffix link of $v$;
        end
    end
end
```

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

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 [23]. 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 context-free grammar (RLCFG), which includes run-length rules of constant size. To simplify, we disallow rules of the form $A \rightarrow \varepsilon$, which are easily removed without increasing the grammar size.

▶ **Definition 4.** *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 $A \rightarrow B_1 \cdots B_k$ for $k > 0$ and terminals or nonterminals B_i (this rule is said to be of size k), and of the form $A \rightarrow B^k$ for $k > 1$ and a terminal or nonterminal B, which is identical to $A \rightarrow B \cdots B$ with k copies of B, but is said to be of size 2. The* size *of the RLCFG is the sum of the sizes of all of its rules. A* Context-Free Grammar (CFG) *for T is a RLCFG for T that does not use rules of the form $A \rightarrow B^k$.*

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 [34]. While finding the smallest grammar is NP-hard [10], heuristics like RePair obtain very good results [27].

Note that our RLCFGs have a unique parse tree, defined as follows [11, Sec. 4].

▶ **Definition 5.** *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 $A \rightarrow B_1 \cdots B_k$, then the node has k children labeled $B_1, \ldots, B_k$ left to right. If its rule is $A \rightarrow B^k$, then the node has k children labeled B. It follows that the ith left-to-right leaf of the parse tree is labeled $T[i]$.*

While the parse tree has size $\Theta(n)$, a convenient representation of a RLCFG is the so-called grammar tree, which is of size $O(g_{rl})$ [11, Sec. 6].

▶ **Definition 6.** *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 $A \rightarrow B^k$ we preserve their first child only, replacing the other $k-1$ children (which are leaves) with a single special leaf labeled $B^{[k-1]}$. 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. We call $exp(A)$ the string of terminals to which symbol A expands, and $exp(a) = a$ for terminals a. The grammar tree defines a parse of T, as follows.

▶ **Definition 7.** *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 *$exp(v_i)$.*

A classic grammar-based index [12] 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 boundary are taken as primary). It uses the fact that every occurrence has primary occurrences and that all the secondary ones can be found inside pruned leaves of nonterminals that contain other occurrences. In this paper we will be interested in the mechanism to find the primary occurrences. This 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 [11, Sec. 6 and App. A].

▶ **Definition 8.** *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 $exp(B)^{t-1}$ (i.e., $t-1$ concatenations of $exp(B)$) 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 $P_{\mathcal{X}}$ storing the strings of $\mathcal{X}$ and another Patricia tree $P_{\mathcal{Y}}$ storing the strings of $\mathcal{Y}$ [32]. Let us add some data to nodes for our convenience. Each Patricia tree node v stores its 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.

Each primary occurrence consists of a suffix of some string $X \in \mathcal{X}$ matching $P[1\,..\,i]$ corresponding to some string $Y \in \mathcal{Y}$ whose prefix matches $P[i+1\,..\,m]$, for some $1 \leq i < m$ (if $m = 1$, it is just a suffix of X matching P) [11, Sec. A.4]. Therefore, to find the primary occurrences of P, the index tries out every cutting point i, and searches $P_{\mathcal{X}}$ for $P[1\,..\,i]^{rev}$ and $P_{\mathcal{Y}}$ for $P[i+1\,..\,m]$. If both nodes $x \in P_{\mathcal{X}}$ and $y \in P_{\mathcal{Y}}$ 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 i, and are efficiently found with a geometric data structure on $\mathcal{G}$. By storing the position t of T where $exp(B_{s-1})$ ends for such point, we know that P occurs in $T[t-i+1\,..\,t-i+m]$ (the actual index stores pointers to the grammar tree, but this suffices for us).

Both the Patricia trees and the grid take $O(g_{rl})$ space. The index also needs to verify the matches of the Patricia trees. It uses an $O(g_{rl})$-space data structure $\mathcal{A}$ that can read, in $O(\ell)$ time, any length-ℓ prefix or suffix of $exp(A)$, for any nonterminal A [11, Lem 6.6]. If x is a node of $P_{\mathcal{X}}$, its corresponding string is the $|x|$-length reversed suffix of any string between the x^1th and the x^2th in $\mathcal{X}$. Let $X = exp(B_{s-1})^{rev}$ be one such string, then we store $\langle v \rangle = B_{s-1}$ associated with v. Similarly, a node $v \in P_{\mathcal{Y}}$ that prefixes $exp(B_s) \cdots exp(B_t)$ stores $\langle v \rangle = B_s$ (from where we can obtain the subsequent siblings). We can then obtain the string represented by any v using $\mathcal{A}$ on $\langle v \rangle$.

4 A Quadratic-Time Solution

We now present a quadratic-time solution 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 any CFG is a particular case of RLCFG, our algorithm also runs with any CFG.

The generic idea follows that of Algorithm 1, sliding a window $P[i\,..\,j]$ along the pattern. We maintain a set of so-called *active positions* $r \in [i\,..\,j]$.

▶ **Definition 9.** *A position $r \in [i\,..\,j]$ is* active *if $P[r+1\,..\,j]$ prefixes some string in $P_{\mathcal{Y}}$.*

Note that, since we slide the window $P[i\,..\,j]$ forwards, once a position r becomes inactive, it will not become active again.

4.1 Algorithm

The algorithm maintains the invariant that, when the window is $P[i\,..\,j]$, (i, j) is the last MEM of $P[1\,..\,j]$ (if $i \leq j$) and all the MEMs ending before j have already been reported. It maintains the set $R \subseteq [i\,..\,j]$ of active positions, and for each such active position $r \in R$:

- The node $y_r \in P_{\mathcal{Y}}$ 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 $P_{\mathcal{Y}}$; note that r is active iff $r + \ell_r \geq j$.
- The node $x_r \in P_{\mathcal{X}}$ corresponding to the longest prefix of $P[i\,..\,r]^{rev}$ that exists in $P_{\mathcal{X}}$, and such that there are points in $\mathcal{G}$ in the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$. 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 $P[r - |x_r| + 1\,..\,r]^{rev}$.

Our algorithm, depicted in Algorithm 2, iterates over j, from 0 to $m-1$, and at each cycle it extends the current window to end 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. Line 3 first sees, in this case, if we can descend from the root of $P_{\mathcal{X}}$ by $P[j+1]$, to start a new nonempty substring $P[j+1, j+1]$. If this is not possible, it just increases i and goes for 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.

Lines 5–7 first create the new active position $r = j+1$, with corresponding y_r set at the root of $P_{\mathcal{Y}}$. To compute ℓ_r, we descend in $P_{\mathcal{Y}}$ as much as possible by $P[r+1\,..]$. To compute x_r, we also descend in $P_{\mathcal{X}}$ 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 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 $P_{\mathcal{Y}}$ (at most $\ell_r + 1$ characters) or the needed suffix in $P_{\mathcal{X}}$ (at most $|x_r| + 1$ characters).

Lines 8–16 then remove the active positions that do not reach $j+1$ and updates the variables for 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 $P_{\mathcal{Y}}$ in line 12 (this shrinks the range $[y_r^1, y_r^2]$). Note that, once we know that we can descend from y_r by $P[j+1]$ (because $r + \ell_r \geq j+1$), 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 that would have been needed to descend from virtual nodes $y_r \in P_{\mathcal{Y}}$: when y_r is not the root, its text position is not phrase-aligned, so we cannot access its first symbols in constant time using $\mathcal{A}$.

Line 13 updates the nodes x_r of the surviving active positions, because some ranges $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ could be empty after we reduce $[y_r^1, y_r^2]$. 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 $P_{\mathcal{X}}$. 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.

Lines 8, 14, and 17 recompute the value $p = \min\{r - |x_r| + 1,\ r \text{ is active}\}$. This is necessary to make i grow as needed so that $P[i\,..\,j+1]$ occurs in T, then reestablishing the invariant that $P[i\,..\,j+1]$ is the last MEM of $P[..\,j+1]$. If $p = i$, then $P[i\,..\,j+1]$ occurs in T (as it has a primary occurrence in some $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$), so we can retain the current value of i; line 18 collects some text position t to be reported in case $(i, j+1)$ turns out to be a MEM of the whole P. If, on the other hand, $p > i$, this means $P[i\,..\,j+1]$ does not occur in T and thus (i, j) was a MEM. Lines 20–21 then report MEM (i, j) with its text position t (collected in the previous cycle of j) and increase i to p, since only $P[p\,..\,j+1]$ occurs in T. This could make i exceed $j+1$ when the window becomes empty; otherwise line 23 finally inserts $j+1$ as an active position. Line 26 reports the final MEM when j reaches m.

```
1  $i \leftarrow 1$; $R \leftarrow \emptyset$;
2  for $j \leftarrow 0, \ldots, m-1$ do
3      if $i = j+1$ and the root of $P_{\mathcal{X}}$ has no child labeled $P[j+1]$ then $i \leftarrow i+1$ ;
4      else
5          $y_{j+1} \leftarrow$ root of $P_{\mathcal{Y}}$;
6          $v \leftarrow$ descend in $P_{\mathcal{Y}}$ as much as possible with $P[j+2\,..]$; $\ell_{j+1} \leftarrow |v|$;
7          $x_{j+1} \leftarrow$ descend in $P_{\mathcal{X}}$ as much as possible with $P[i\,..\,j+1]^{rev}$;
8          $r_{\min} \leftarrow j+1$;
9          for $r \in R$ do
10             if $r + \ell_r = j$ then $R \leftarrow R \setminus \{r\}$ ;
11             else
12                 $y_r \leftarrow$ child of $y_r$ by $P[j+1]$;
13                 while the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ is empty do $x_r \leftarrow$ parent of $x_r$ ;
14                 if $r - |x_r| < r_{\min} - |x_{r_{\min}}|$ then $r_{\min} \leftarrow r$ ;
15             end
16         end
17         $p \leftarrow r_{\min} - |x_{r_{\min}}| + 1$;
18         if $p = i$ then $t \leftarrow$ text position of some point in $[x^1_{r_{\min}}, x^2_{r_{\min}}] \times [y^1_{r_{\min}}, y^2_{r_{\min}}]$ ;
19         else
20             report $(i,j)$ with position $T[t-j+i\,..\,t]$;
21             $i \leftarrow p$
22         end
23         if $i \leq j+1$ then $R \leftarrow R \cup \{j+1\}$ ;
24     end
25 end
26 if $i \leq m$ then report $(i,m)$ with position $T[t-m+i\,..\,t]$;
```

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

4.2 Analysis

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

The costs of lines 6, 7, and 13, 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 phyisical ancestor of x_r rather than to its possibly virtual parent node, since 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. Maintaining the variables associated with active positions allows us amortizing these costs along the process.

Emptiness queries on $\mathcal{G}$ can be solved in $O(\log^{\epsilon} g_{rl})$ time and $O(g_{rl})$ space for any constant $\epsilon > 0$ [9]; a recent construction takes $O(g\sqrt{\log g})$ time [2]. The same complexity holds for returning one point in nonempty ranges. The $O(m^2)$ cost charged to positions r is then multiplied by this factor. The rest of the construction time is inherited from the CFG-based index [12]; extending it to RLCFGs does not increase it.

▶ **Theorem 10.** *Let g_{rl} be the size of a RLCFG generating only $T[1\,..\,n]$. Then, for any* *constant $\epsilon > 0$, we can build in $O(g_{rl} \log^2 n)$ time a data structure of size $O(g_{rl})$ that finds the MEMs of any given pattern $P[1\,..\,m]$ in time $O(m^2 \log^\epsilon g_{rl}) \subseteq O(m^2(\log^\epsilon \delta + \log\log n))$, with an occurrence of each. The query process uses $O(m)$ additional space.*

As mentioned, any CFG can also be used in the theorem. By using an emptiness structure of size $O(g_{rl} \log\log g_{rl})$ [9], we find the MEMs in time $O(m^2 \log\log g_{rl})$.

5 Indexing Locally Consistent Grammars

Before entering into the details of our more sophisticated solution, we must introduce some new concepts. A *locally consistent grammar* 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. This has 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 [11, 25, 24]. In this paper we make use of the latest result [24]. We present a lighter informal description; see the original paper for full details.

5.1 The Grammar

We first define the grammar [24, Sec. 3], which is produced level by level, for $O(\log n)$ levels. 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 such that $|exp(A)| \le \ell_k$. Those are the symbols that can be grouped to form new nonterminals in level k.

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 $\mathcal{A}_1 = \Sigma$ (Σ is the alphabet of T and also the set of terminals of the RLCFG). For each such run, say of t symbols $a \in \mathcal{A}_1$, we create the rule $A \rightarrow a^t$ and replace the run by the nonterminal A. The resulting sequence after all the runs have been replaced is $S_1 = rle_{\mathcal{A}_1}(S_0)$, 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 position $0 < i < |S_1|$ such that

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

is the end of 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 $A \rightarrow S_1[i\,..\,j]$ and replace every occurrence of the same block in S_1 by A. The resulting string is called $S_2 = bc_{\pi_2,\mathcal{A}_2}(S_1)$. The process continues in the same way for odd and even levels:

$$\begin{aligned} S_k = &\quad rle_{\mathcal{A}_k}(S_{k-1}) && \text{if } k \text{ is odd,} \\ S_k = &\quad bc_{\pi_k,\mathcal{A}_k}(S_{k-1}) && \text{if } k \text{ is even,} \end{aligned}$$

until we reach $|S_k| = 1$ for some $k = O(\log n)$. The algorithm is Las Vegas type, trying out functions π_k to obtain some desired grammar size, but otherwise any functions π_k yield a correct index. They [24] prove that, in $O(n)$ expected time, a RLCFG of size $O(\delta \log \frac{n}{\delta})$ is obtained, where δ is a lower bound measure based on the substring complexity of T [11]: let T_ℓ be the number of distinct length-ℓ substrings in T, then $\delta = \max\{T_\ell/\ell, \ell > 0\}$. Interestingly, for every n and δ, there exists a string family that *requires* $\Omega(\delta \log \frac{n}{\delta})$ space (i.e., $\log(n)$-bit words) to be represented [25]; therefore using space $O(\delta \log \frac{n}{\delta})$ for a grammar (and for an index) is asymptotically optimal for any specific n and δ.

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. Concretely, 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 [24, 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

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

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

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

The key point [24] 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, $|M(P)| = O(\log m)$. 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 A Faster Solution using Locally Consistent Grammars

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 it through P, and use it 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.

6.1 Parsing the Pattern

In this section we show how we maintain the parse tree of $P[i\,..\,j]$, or more precisely, the corresponding strings $S_0, S_1, \ldots$, as well as $M(P[i\,..\,j])$ and R, as we slide $P[i\,..\,j]$ along P. Recall that the height of the parse tree of P is $O(\log m)$ in our locally-balanced grammar.

Maintaining the parse

Assume the parse tree is built for $P[i\,..\,j]$ and now we have to increment j. At level $k = 0$, we simply extend S_0 by the symbol $e_0 = P[j+1]$. This propagates upwards as follows, where l_k is the last symbol of S_k and e_{k-1} has just been added at the end of S_{k-1}.

1. If k is odd (a run-formation level), $l_k = l_{k-1}$ or $l_k \to l_{k-1}^t$, $|exp(e_{k-1})| \leq \ell_k$, and $l_{k-1} = e_{k-1}$, we find or create a rule $e_k \to l_{k-1}^{t+1}$ ($t = 1$ if $l_k = l_{k-1}$) and replace l_k by e_k.
2. If k is even (a block-formation level), $l_k = l_{k-1}$ or $l_k \to \beta l_{k-1}$, $|exp(e_{k-1})| \leq \ell_k$, and $\pi_k(l_{k-1}) > \pi_k(e_{k-1})$, we find or create a rule $e_k \to \beta l_{k-1} e_{k-1}$ ($\beta = \varepsilon$ if $l_k = l_{k-1}$) and replace l_k by e_k.
3. In any other case, we just append $e_k = e_{k-1}$ at the end of S_k.

We see that insertions at the end of S_{k-1} propagate as new insertions or replacements at the end of S_k. We process those replacements as the deletion of l_k followed by the insertion of e_k at the end. The following are the rules to propagate to S_k the deletion of l_{k-1}.

1. If $l_k = l_{k-1}$, we delete l_k.
2. If $l_k \to l_{k-1}^t$ is a run, we find or create the rule $e_k \to l_{k-1}^{t-1}$ (just $e_k = l_{k-1}$ if $t-1 = 1$) and replace l_k by e_k.
3. If $l_k \to \beta\, l_{k-1}$ is a block, we find or create the rule $e_k \to \beta$ (just $e_k = \beta$ if $|\beta| = 1$) and replace l_k by e_k.

Each of those updates can be carried out in constant time by just maintaining linked lists with the sequence of symbols in each string S_k, perfect hash tables with the existing right-hand sides of the run-formation rules $l_k \to l_{k-1}^t$, and tries with the right-hand sides of the block-formation rules $l_k \to \beta\, l_{k-1}$. In particular, each block-formation nonterminal l_k points to the node in the trie that represents its string $\beta\, l_{k-1}$, and the trie node representing $\beta\, l_{k-1}$ stores l_k. With parent pointers in the trie, we have constant-time access to the node of β from the node of $\beta\, l_{k-1}$, and with child pointers we move from $\beta\, l_{k-1}$ to $\beta\, l_{k-1}\, e_{k-1}$. The children of trie nodes are stored in perfect hash tables to enable downward traversals in constant time. All this can be precomputed in expected time linear in the grammar size.

The case when i increases is symmetric. We start by deleting the first symbol of S_0, and propagate the update upwards acting on the first symbols f_k at each level k. To handle those operations we need the lists for S_k be doubly-linked, and also to store tries for all the right-hand sides read as $f_k \to f_{k-1}\beta'$.

The number of updatess are actually bounded to $O(1)$ updates per level, and thus to $O(\log m)$ per increase of j or of i. Consider the string $P[i\,..\,j] \cdot \$$, where \$ is a special symbol for which we assume $\pi_k(\$) = +\infty$ for all k. The parse tree of $P[i\,..\,j] \cdot \$$ is then identical to that of $P[i\,..\,j]$, just adding \$ at the end of every S_k. The strings S_k formed for $P[i\,..\,j]$ followed by \$ are the same formed for $P[i\,..\,j]$ followed by $P[j+1]$, except for the first $2\alpha_k$ and the last α_k positions of $P[i\,..\,j]$ [24, Lem. 3.7]. Therefore, the addition of $P[j+1]$ can only alter the last $1+\alpha_k$ positions of $P[i\,..\,j+1]$ in each S_k. Analogously, removing $P[i]$ can only alter the first $2\alpha_k - 1$ positions of $P[i+1\,..\,j]$ in its strings S_k. On the other hand, those $O(\alpha_k)$ positions correspond to only $O(1)$ symbols in S_k [24, Lem. 3.8]. The total amount of work is proportional to the number of updated symbols if we perform them levelwise, on S_0, then on S_1, and so on. All the changes then add up to $O(m \log m)$ along the processing of P.

Dealing with unknown symbols

We analyzed the parsing process as if we would always find a known nonterminal for the right-hand sides we modify, but it could be that we have to create new nonterminals that were never formed during the parsing of T.

To handle those cases, we create fresh nonterminals and continue the process normally, removing them when they are no longer needed. An easy way to handle this would be to make the hash tables and tries of right-hand sides dynamic, so we can add and remove elements in the tables and nodes; we will soon sharpen this solution.

We must assign values $\pi_k(e_k)$ to the fresh symbols e_k we create in S_k. We can assign arbitrary values (different from the other π_k values) without affecting correctness: the index works correctly for arbitrary functions π_k, as explained. No matter how we choose the values $\pi_k(e_k)$, we can add $O(m \log m)$ fresh symbols along the whole process, but we can do better.

New symbols e_k may appear in the parsing of $P[i \mathinner{.\,.} j]$ even if $P[i \mathinner{.\,.} j]$ appears in T, because the parsing of $P[i \mathinner{.\,.} j]$ can be different from that of its occurrences in T. However, this can happen only in the first $2\alpha_k$ and the last α_k positions, in S_k. Once the end of e_k falls before position $j - \alpha_k$, and it is after the position $2\alpha_k$, then e_k should have appeared in the parsing of every occurrence of $P[i \mathinner{.\,.} j]$ in T [24, Lem 3.7]. Therefore, when we completely incorporate $P[j+1]$ and as a result the end of a fresh symbol e_k of S_k falls behind position $j+1-\alpha_k$ of $P[i \mathinner{.\,.} j+1]$, we know $P[i \mathinner{.\,.} j+1]$ cannot appear in T until the position falls behind $i + 2\alpha_k$. At this point, then, we can suspend the search (very much as Algorithm 2 does in line 3) and increase i until e_k ends within the leftmost $2\alpha_k$ symbols of $P[i \mathinner{.\,.} j+1]$.

This has as a consequence that we can have only $O(1)$ fresh symbols per level, just as $M_{i,j}(P)$, and $O(\log m)$ in total. Instead of making the tries and hash tables dynamic, we can have one extra atomic heap per hash table (the one for the run-length symbols and the one in each trie node) where we can insert/delete the necessary fresh symbols, and they will be processed in constant time. We then retain the $O(m \log m)$ total processing cost.

Maintaining $M(P)$ and R

After we finish updating the parse tree, we collect the first $2\alpha_{k+1}$ positions, the position of the following end of block, and the last α_{k+1} positions, in each list S_k, to form the sets $M_k(P)$. Those are then merged into $M(P)$ and sorted by increasing value. Since $|M(P)| = O(\log m)$, and its values are integers in $[1 \mathinner{.\,.} m]$, $M(P)$ can be sorted in $O(\log m)$ time with atomic heaps. We then traverse $M(P)$ and the current set R in synchronization, so as to (1) remove the positions of R that are not anymore in $M(P)$, and (2) insert in R the positions that are now in $M(P)$, as long as the position had not been in R before and had been inactivated (this is easily marked in an array of size m). At the end of this process, it always holds that $R \subseteq M(P[i \mathinner{.\,.} j])$, and thus $|R| = O(\log m)$. Due to the parsing, an active position r may enter and leave R several times along the process, but this time that will not be an issue.

Overall, we maintain the parsing, $M(P[i \mathinner{.\,.} j])$, and R in time $O(m \log m)$. Since all the lines in Algorithm 2 other than 6, 7, and 13, take $O(1)$ time per element in R, the total time spent in those lines adds up to $O(m \log m)$.

6.2 Patricia Tree Searches

Lines 6 and 7 of Algorithm 2 perform $\Theta(m)$-time searches in $P_{\mathcal{X}}$ and $P_{\mathcal{Y}}$. Since each of the m positions becomes active when $j+1$ reaches it, this amortizes to no less than $\Theta(m^2)$, which is now too high for us. We then resort to a different technique.

Instead of computing ℓ_{j+1} and x_{j+1} inside the main cycle, we will compute them all beforehand, in batch form. We make use of the following result, which was key to obtain subquadratic times in grammar-based indexing.

▶ **Lemma 11.** *([11, 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 S in time $f_h(\ell)$. We can then build a data structure of $O(|\mathcal{S}|)$ words 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.*

When $\mathcal{S} = \mathcal{X}$ or $\mathcal{S} = \mathcal{Y}$, our access data structure $\mathcal{A}$ provides the required prefix/suffix extraction in time $f_e(\ell) = O(\ell)$. As for Karp-Rabin signatures [20], a result of independent interest is that we can obtain $f_h(\ell) = O(\log \ell)$ time on our grammar, as proved next. We consider the more complicated case of $Y \in \mathcal{Y}$; the case of $X \in \mathcal{X}$ is analogous. Recall we can compute in $O(1)$ time one of $\kappa(S \cdot S')$, $\kappa(S)$, and $\kappa(S')$, from the other two [11, Sec. A.3].

▶ **Lemma 12.** *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 $\mathcal{A}$ used for extraction from the root of $P_\mathcal{Y}$. The strings in $\mathcal{Y}$ are concatenations $Y = exp(B_s) \cdots exp(B_t)$ of siblings in rules $A \rightarrow B_1 \cdots B_t$ in the grammar tree. The node $v \in P_\mathcal{Y}$ 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 $\mathcal{A}$ is a set of tries on the grammar symbols. The terminals Σ form the trie roots. If $A \rightarrow B_1 \cdots B_t$, then B_1 is the parent of A. If $A \rightarrow B^t$, then B is the parent of A. Any ancestor C of B_s in the tries is a node that descends from B_s by the leftmost path in the parse tree. The structure $\mathcal{A}$ can jump from B_s to any such C in constant time in the tries. Our grammar is locally balanced: there can be only one block ending inside $exp(B_s[\,..\,\ell])$ at level $k = 1 + 2\log_{4/3}(4\ell)$ [24, Lem. 3.8], and thus the lowest C such that $|exp(C)| \geq \ell$ has level at most $k+1$; its height is $d \leq k+1 = O(\log \ell)$. It can then be found in $O(\log\log \ell)$ time with exponential search on the ancestors of B_s. We then have that $exp(B_s)[\,..\,\ell] = exp(C)[\,..\,\ell]$ and can compute the signature on C instead.

The basic algorithm to compute signatures takes time $O(\log^2 \ell)$ [11, Lem 6.7]. It moves from C towards the leaf L of the parse tree that corresponds to $exp(C)[\ell]$. Let $C \rightarrow C_1 \cdots C_t$, 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 $C \rightarrow C_1^t$ 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 [11, Thm. A.3] 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 $C \rightarrow C_1^t$ 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 children C_i of C belonged 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 [11, Lem 6.7] and then continue from C_i, where the more sophisticated technique [11, 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 $\kappa(Y[\,..\,\ell]) = \kappa(exp(A)[w_{s-1}(A) + 1\,..\,w_{s-1}(A) + \ell])$. ◀

Batched searches

The m searches for all the values ℓ_r, $1 \leq r \leq m$, correspond to searching $P_{\mathcal{Y}}$ for every suffix $P[r+1\,..]$. Note that Lemma 11 does not yield the node v of line 6, but rather its corresponding range $[v^1, v^2]$. By performing a lowest common ancestor (LCA) query on $P_{\mathcal{Y}}$ from the v^1th and v^2th leaves, we obtain $v = lca(v^1, v^2)$ (identifying leaves with their ranks). The answer is indeed v if $|v| = m - r$; if $m - r < |v|$ the answer is the virtual node of string length $m - r$ on the edge of $P_{\mathcal{Y}}$ that leads to v. Linear-space LCA data structures that are built in linear time and answer lca in $O(1)$ time are well known [3].

The problem is that Lemma 11 works only if $P[r+1\,..]$ actually prefixes some string in $\mathcal{Y}$. Otherwise, unlike classical trie searching, it does not even 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 11 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 be a prefix in $\mathcal{Y}$ and $P[r+1\,..\,b+1]$ is known not to be a prefix in $\mathcal{Y}$ (the first condition is assumed to hold if $r+1 > a$). 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 11, with $c = (a+b+1)/2$. 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 $r+1 > c$, then $P[r+1\,..\,c] = \varepsilon$, so we can directly move r to $\mathcal{Q}_{c,b}$ without searching for it). We will associate the node $v_{r,c} \in P_{\mathcal{Y}}$ to those values r for which $P[r+1\,..\,c]$ is found in $\mathcal{Y}$; those not found retain their previous node $v_{r,*}$ (in the beginning all such nodes are $v_{r,r}$ and equal to the root of $P_{\mathcal{Y}}$).

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 $v_{r,c}$.

The cost of using Lemma 11 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 positions r. The second part is potentially very large, however: the suffixes in $\mathcal{Q}_{a,b}$ may start well ahead 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.

Smarter substring extractions

To reduce this time, we consider where the $O(m)$ cost in Lemma 11 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 $1 \leq r \leq m$, 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 $P'[\,..\,c-r]$ regularly in time $f_e(c-r) = O(b-a)$ with $\mathcal{A}$, because P' starts at the root of $P_{\mathcal{Y}}$ and thus it belongs to $\mathcal{Y}$.

Otherwise, $r+1 \leq a$ and thus $P[r+1\,..\,a]$ had been successfully matched before and we have its node $v_{r,a} \in P_{\mathcal{Y}}$. As mentioned, the process of Lemma 11 performs several checks before performing 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 $P'[..\,c-r]$ does start with $P[r+1\,..\,a]$, and then only need to extract $P'[a-r+1\,..\,c-r]$ from the text, which is of length $O(b-a)$.

This time, however, the string to extract does not start at the root of $P_{\mathcal{Y}}$, and thus it requires a random access to T.[1] Recall, as in Lemma 12, that the strings in $\mathcal{Y}$ are concatenations $Y = exp(B_s)\cdots exp(B_t)$ of consecutive siblings in rules $A \rightarrow B_1 \cdots B_t$ in the grammar tree (if $A \rightarrow B^t$, then the node stores $\langle v \rangle = B^{[t-1]}$ and we have $B_s = B$). Let us first assume that $|exp(B_s)| \geq c$, so the substring to extract is within $exp(B_s)[..\,c]$. We use again the structure $\mathcal{A}$, now to extract the string in time $O(b-a+\log c)$.

Once again, we can search in $O(\log\log c)$ time for the lowest descendant C of B_s such that $|exp(C)| \geq c$; its height is $d = O(\log c)$ because the grammar is locally balanced. Since $exp(B_s)[..\,c] = exp(C)[..\,c]$, 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 12, the time is $O(d) = O(\log c)$. From L, $exp(C)[a-r+1\,..\,c-r] = P'[a-r+1\,..\,c-r]$ is extracted in time $O(c-a) = O(b-a)$.

In case $|exp(B_s)| < c$, 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 $O(\log c) = O(\log m)$ cost can be charged to the suffix sought, which adds up to $O(m \log^2 m)$ over the $O(\log m)$ times each suffix may use the lemma. The $O(b-a)$ terms add up to $O(m)$ per level of sets $Q_{a,b}$ (a level corresponding to a difference $b-a+1$). Since all the ranges (a,b) of a level are disjoint (level ℓ partitions $(1,m)$ into 2^ℓ ranges of size $m/2^\ell$), the $b-a$ values add up to $O(m)$ per level. Since there are $O(\log m)$ levels, that part of the cost adds up to $O(m \log m)$.

We similarly compute the nodes x_r for every $P[..\,r]^{rev}$ on $P_{\mathcal{X}}$. While in line 7 of Algorithm 2 we search only for $P[i\,..\,r]^{rev}$ because we are not interested in positions before i, this time we precompute all the values in advance for the smallest $i = 1$. Later, when i increases, we will move to the required ancestors of x_r in line 13.

Overall, the Patricia trie searches execute lines 6 and 7 in $O(m \log^2 m)$ total time.

6.3 Emptiness Queries

In line 13, we perform one range emptiness query every time we decrement $|x_r|$ for some r; this amounts to $O(m^2)$ emptiness queries, which we cannot afford now. We will instead use a faster method based on orthogonal range successor queries: given a range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$, we can find the largest value $x_< \leq x_r^1$ such that $[x_<, x_r^2] \times [y_r^1, y_r^2]$ contains a point, and the smallest value $x_> \geq x_r^2$ such that $[x_r^1, x_>] \times [y_r^1, y_r^2]$ contains a point. Those queries can 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 [35]; construction time can be made $O(g\sqrt{\log g})$ [2].

The lowest ancestor x of x_r containing some point in $[x^1, x^2] \times [y_r^1, y_r^2]$ must then hold $x^1 \leq x_<$ or $x^2 \geq x_>$. In the first case, it is $v_1 = lca(x_<, x_2)$; in the second, it is $v_2 = lca(x_1, x_>)$. Both v_1 and v_2 are ancestors of x_r, and thus of each other. The correct node x is then the lowest of v_1 and v_2, which is known from the leaf ranges stored at the nodes.

[1] It is tempting to say that, since we had already matched $P[r+1\,..\,a]$ from the root of $P_{\mathcal{Y}}$, 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 longer suffix, starting before $r+1$.

With this query, line 17 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.

6.4 The Final Result

Our time complexities then add up to $O(m \log m(\log m + \log^\epsilon g))$ for a grammar of size g. Since in our case $g = O(\delta \log \frac{n}{\delta})$, 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 11 [11, Sec. 6.6] (the grammar is built in $O(n)$ expected time, see [24, Cor. 3.15]).

▶ **Theorem 13.** *Let $T[1\,..\,n]$ have substring complexity δ. Then, for any constant $\epsilon > 0$, we can build in $O(n \log n)$ expected time a data structure of size $O(\delta \log \frac{n}{\delta})$ that finds the MEMs of any given pattern $P[1\,..\,m]$ in time $O(m \log m(\log m + \log^\epsilon \delta + \log\log n)) \subseteq O(m \log m(\log m + \log^\epsilon n))$, with an occurrence of each. The query process uses $O(m)$ additional space.*

We have assumed $m \le n$, but it could be the other way in some applications. Since in this case no substring longer than n will be matched inside P, we can run $O(m/n)$ iterations finding the MEMs of $P[1\,..\,2n]$, $P[n\,..\,3n]$, $P[2n\,..\,4n]$, and so on, avoiding repeated MEMs in the output. The total cost would then be $O(m \log^2 n)$ and the query space would be $O(n)$.

7 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 [34]. 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 [34]. 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 [18], which could also run in $O(g_{rl})$ space. Within $O(\delta \log \frac{n}{\delta})$ 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 and δ, though g_{rl} is always $O(\delta \log \frac{n}{\delta})$ and can be $o(\delta \log \frac{n}{\delta})$ in some text families [25].

A challenge for future work is to extend our results to finding k-MEMs, which are the maximal substrings of P that appear at least k times in T, for k given at query time. The basic Algorithm 1 is easily modified to find the k-MEMs in $O(m)$ time, but running in compressed space is more costly. In the extended version we will show how find k-MEMs, changing the $\log^\epsilon n$ terms to $\log^{2+\epsilon} n$, by building on grammar-based indices that can count the number of occurrences associated with a set of primary occurrences [11]. The space also increases: the $O(g_{rl})$ space becomes $O(g)$ ($g \ge g_{rl}$ being size of the smallest CFG) and the $O(\delta \log \frac{n}{\delta})$ space becomes $O(\gamma \log \frac{n}{\gamma})$ ($\gamma \ge \delta$ being the size of a string attractor of T [22]).

Our techniques are presented on a particular locally consistent grammar [24] that yields the best complexities, but they would work on others too. We plan to implement them on practical constructions of CFGs [12] built with RePair [27] or of locally consistent grammars based on induced suffix sorting [14, 36]. Further, even without having theoretical guarantees, the algorithm for arbitrary RLCFGs will probably be competitive if implemented on Lempel-Ziv based indices [26, 16], which are considerably smaller than those based on grammars.

References

1 H. Bannai, T. Gagie, and T. I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

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

3 M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

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

5 C. Boucher, O. Cvacho, T. Gagie, J. Holub G. Manzini, G. Navarro, and M. Rossi. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–72, 2021.

6 C. Boucher, T. Gagie, T. I, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. 31th Data Compression Conference (DCC)*, pages 193–202, 2021.

7 M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

8 M. Cáceres and G. Navarro. Faster repetition-aware compressed suffix trees based on block trees. *Information and Computation*, 285B:article 104749, 2022.

9 T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

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

11 A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):article 8, 2020.

12 F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

13 M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

14 D. Díaz-Domínguez, G. Navarro, and A. Pacheco. An LMS-based grammar self-index with local consistency properties. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 100–113, 2021.

15 A. Farruggia, T. Gagie, G. Navarro, S. J. Puglisi, and J. Sirén. Relative suffix trees. *The Computer Journal*, 61(5):773–788, 2018.

16 H. Ferrada, D. Kempa, and S. J. Puglisi. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8, 2018.

17 T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

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

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

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

21 D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020.

22 D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 827–840, 2018.

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

24 T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ-optimal space. In *Proc. 15th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–103, 2022.

25 T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023.

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

27 J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

28 V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.

29 V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

30 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

31 E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

32 D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

33 G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.

34 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

35 Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*, pages 271–282, 2012.

36 D. Nunes, F. Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. Grammar compression by induced suffix sorting. *ACM Journal of Experimental Algorithmics*, 27:article 1.1, 2022.

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

38 M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.

39 L. M. S. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.

40 I. Tatarnikov, A. S. Farahani, S. Kashgouli, and T. Gagie. MONI can find k-MEMs. *CoRR*, 2202.05085, 2022.

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