Theory of Computing Systems (2026) 70:11
https://doi.org/10.1007/s00224-026-10266-x

Logarithmic-Time Internal Pattern Matching Queries in Compressed and Dynamic Texts

Anouk Duyster[1,2] · **Tomasz Kociumaka**[1]

Received: 17 March 2025 / Accepted: 2 February 2026
© The Author(s) 2026

Abstract
Internal Pattern Matching (IPM) queries on a length-n text T, given two fragments X and Y of T such that $|Y| < 2|X|$, ask to compute all exact occurrences of X within Y. IPM queries have been introduced by Kociumaka, Radoszewski, Rytter, and Waleń [SODA'15 & SICOMP'24], who showed that they can be answered in $\mathcal{O}(1)$ time using a data structure of size $\mathcal{O}(n)$ and used this result to answer various queries about fragments of T. In this work, we study IPM queries on compressed and dynamic strings. Our result is an $\mathcal{O}(\log n)$-time query algorithm applicable to any balanced recompression-based run-length straight-line program (RLSLP). In particular, one can use it on top of the RLSLP of Kociumaka, Navarro, and Prezza [IEEE TIT'23], whose size $\mathcal{O}\big(\delta \log \frac{n \log \sigma}{\delta \log n}\big)$ is optimal (among all text representations) as a function of the text length n, the alphabet size σ, and the substring complexity δ. Our procedure does not rely on any preprocessing of the underlying RLSLP, which makes it readily applicable on top of the dynamic strings data structure of Gawrychowski, Karczmarz, Kociumaka, Łącki and Sankowski [SODA'18], which supports fully persistent updates in logarithmic time with high probability.

Keywords Internal pattern matching · Recompression · Text indexing

1 Introduction

Given two fragments $X = T[x..x')$ and $Y = T[y..y')$ of a text T such that $|Y| < 2|X|$,[1] an Internal Pattern Matching (IPM) query reports all exact occurrences

[1] The restriction $|Y| < 2|X|$ ensures that the starting positions of the reported fragments form an arithmetic progression and thus can be represented in constant space.

✉ Anouk Duyster
aduyster@mpi-sws.org

Tomasz Kociumaka
tomasz.kociumaka@mpi-inf.mpg.de

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

[2] Saarbrücken Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Published online: 23 February 2026

of X within Y, i.e., all fragments matching X and contained within Y. Kociumaka et al. [1] introduced IPM queries as a central building block for answering various further queries about fragments of T. They showed that, for every text of length n, IPM queries can be answered in $\mathcal{O}(1)$ time using a data structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n)$ expected time. The journal version of their work [2] provides an improved data structure that takes $\mathcal{O}(n/\log_\sigma n)$ space and can be deterministically constructed in $\mathcal{O}(n/\log_\sigma n)$ time if the characters of T are integers in $[0\,..\,\sigma)$ for some $\sigma = n^{\mathcal{O}(1)}$. Hence, in the standard setting, IPM queries admit a compact data structure with optimal query and construction time.

Due to their multiple applications (see [2, Sections 1.2–1.3]), IPM queries have also been considered in further settings, including the compressed setting, where the goal is to exploit compressibility of the text, and the dynamic setting, where the text may change over time. Along with the Longest Common Extension (LCE) queries [3], IPM queries constitute an elementary operation of the PILLAR model, introduced by Charalampopoulos et al. [4] with the aim of unifying approximate pattern-matching algorithms across different settings. Whereas LCE queries can be answered in $\mathcal{O}(\log n)$ time in the compressed [5, 6] and dynamic settings [7, 8], the algorithms for IPM queries have been slower so far. In this work, we propose a novel procedure that answers IPM queries in $\mathcal{O}(\log n)$ time in the compressed and dynamic settings, and thus we eliminate the bottleneck in the state-of-the-art PILLAR model implementations and multiple approximate pattern matching algorithms [4, 9–12].

Our Result Efficient implementations of LCE and IPM queries represent compressed and dynamic texts using a run-length straight-line program (RLSLP) constructed using a locally consistent parsing scheme. Such a scheme repeatedly partitions the text into blocks and replaces blocks with individual symbols (so that identical blocks are replaced by the same symbol) until the text consists of a single symbol. The schemes alternate between run-length encoding (with blocks consisting of multiple copies of the same symbol) in odd rounds and another partitioning method (that reduces the text length by a constant factor while ensuring that matching fragments are partitioned consistently, except for blocks at the endpoints) in even rounds. Previous works use many distinct locally consistent parsings. The two most common types are signature encoding by Sahinalp and Vishkin [13, 14], applied in [15–18], and recompression by Jeż [19, 20], applied in [5–7], but there are also a few other schemes; see, e.g., Mehlhorn et al. [15] and Cardinal and Iacono [8].

Our approach relies on *recompression*, which is already known to support $\mathcal{O}(\log n)$-time LCE queries in compressed and dynamic texts [5–7]. In even rounds, this scheme classifies the symbols into *left* and *right* symbols and creates a length-two block out of every left symbol followed by a right symbol; the remaining blocks are of length one.

In this work, we show how to efficiently answer IPM queries using any RLSLP obtained via recompression. Our query algorithm is also compatible with *restricted* recompression [2, 21] that further classifies some symbols as *inactive*; see Definition 8.

Theorem 1 *IPM queries on a text T represented using a (restricted) r-round recompression run-length straight-line program can be answered in $\mathcal{O}(r)$ time.*

We emphasize that our IPM query algorithm does not need any processing of the recompression RLSLP; it only assumes that every non-terminal stores its production, the length of its expansion, and in which round it has been created. Consequently, our approach is automatically applicable on top of every existing construction of a recompression RLSLP or a restricted recompression RLSLP, and our IPM functionality comes at no extra cost in terms of space complexity or construction time.

IPM Queries in Compressed Texts Kociumaka et al. [21] proved that every non-empty text T admits an $\mathcal{O}(\log n)$-round restricted recompression RLSLP of size $\mathcal{O}\big(\delta \log \frac{n \log \sigma}{\delta \log n}\big)$. Here, n is the length of the text, σ is the alphabet size, and δ is the substring complexity of T, defined as $\delta := \max_{1 \le \ell \le n} \frac{d_\ell(T)}{\ell}$, where $d_\ell(T)$ is the number of distinct length-ℓ substrings of T. More recently, Kempa and Kociumaka [6] showed how to construct such an RLSLP in $\mathcal{O}(\delta \log^7 n)$ time from the Lempel–Ziv (LZ77) parsing of T. Combining this with Theorem 1, we obtain the following:

Corollary 2 *For every non-empty text T, there is a data structure of size $\mathcal{O}\big(\delta \log \frac{n \log \sigma}{\delta \log n}\big)$ that answers IPM queries in $\mathcal{O}(\log n)$ time. Such a data structure can be constructed in $\mathcal{O}(\delta \log^7 n)$ time given the LZ77 parsing of T.*

Analogous results are already known for LCE queries [6, Theorem 5.25], which means that the PILLAR model can be implemented in $\mathcal{O}(\log n)$ time per operation using $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ space and $\mathcal{O}(\delta \log^7 n)$ preprocessing time.[2] Prior to this work, state-of-the-art implementations of IPM queries in the compressed setting required $\mathcal{O}(\log^3 n)$ time [22] or $\mathcal{O}(\log^2 n \log \log n)$ time [4] per query.

IPM Queries in Dynamic Texts The dynamic strings data structure of Gawrychowski et al. [7] maintains a collection of non-empty strings, allowing one to add new strings to the collection explicitly (`make_string`), by non-destructively concatenating two existing strings (`concat`), or by non-destructively splitting an existing string into two pieces (`split`). These updates take $\mathcal{O}(\log N)$ time with high probability, where N is the total length of the strings of the collection, except for `make_string`, which takes $\mathcal{O}(n + \log N)$ time for creating a string of length n. Since the dynamic strings data structure of [7] is fully persistent, its space complexity can only be bounded by the total update time (since initialization). Internally, the strings in [7] are represented using an r-round (non-restricted) recompression RLSLP, where $r = \mathcal{O}(\log N)$ with high probability. Hence, Theorem 1 yields the following:

Corollary 3 *The dynamic strings data structure of [7] supports IPM queries in $\mathcal{O}(\log N)$ time with high probability, where N is the total length of the stored strings.*

The dynamic strings data structure of [7] supports LCE queries in $\mathcal{O}(\log N)$ time with high probability, so this yields a PILLAR model implementation in $\mathcal{O}(\log N)$ time

[2] Using the results of [5] instead of [6], one can achieve a faster $\mathcal{O}(g \log \frac{n}{g})$-time construction from a size-g straight-line grammar generating T at the expense of a larger data structure of size $\mathcal{O}(z \log \frac{n}{z})$, where z is the size of the LZ77 parsing of T.

per operation w.h.p. Previous implementations supported IPM queries in $\mathcal{O}(\log^2 N)$ time [4]. Faster IPM queries in the dynamic setting were only known for an alternative deterministic dynamic strings implementation [18] (using a restricted version of signature encoding), which takes $\mathcal{O}(\log N \log^{2-o(1)} \log N)$ time per update as well as for both LCE and IPM queries; see also [10] for a discussion of this alternative.

Conference Version A preliminary version of this work was presented at SPIRE 2024 [23]. The current version also covers our new proof-of-concept Rust implementation (Section 5). Moreover, it contains several simplifications and describes our algorithm in more detail.

2 Preliminaries

In this section, we formally define grammar compression [24], recompression [19, 20], restricted recompression [2, 21], and the underlying concepts and notations. Compared to previous works [2, 6, 7] and the preliminary version of this paper [23], we provide much more details on how to efficiently traverse (uncompacted) parse trees.

2.1 Basic Definitions

A string is a finite sequence of characters from a given alphabet Σ. The length of a string S is denoted $|S|$, and the empty string is denoted ε. For a *position* $i \in [0\,..\,|S|)$,[3] the ith character of S is denoted $S[i]$. A string U is a *substring* of a string S if $U = S[i]S[i+1]\cdots S[j-1]$ holds for some integers $0 \le i \le j \le |S|$. In this case, we say that U *occurs* in S at position i, and we denote the *occurrence* of U at position i of S by $S[i\,..\,j)$. We call $S[i\,..\,j)$ a *fragment* of S and, in some contexts, also denote it by $S[i\,..\,j-1]$ or $S(i-1\,..\,j-1]$. Fragments of the form $S[0\,..\,j)$ and $S[i\,..\,|S|)$ are *prefixes* and *suffixes* of S, respectively.

Technically, a fragment $S[i\,..\,j)$ is interpreted as a tuple (S, i, j) consisting of the positions i and j as well as (a constant reference to) the string S. We say that fragments $S[i\,..\,j)$ and $T[i'\,..\,j')$ of strings S and T, respectively, *match* if they are occurrences of the same substring. Consistently with the literature, we then write $S[i\,..\,j) = T[i'\,..\,j')$ even though the triples (S, i, j) and (T, i', j') are not necessarily equal.

We denote by UV or $U \cdot V$ the concatenation of two strings U and V, that is, $UV = U[0]\cdots U[|U|-1]V[0]\cdots V[|V|-1]$. Furthermore, both $S_0 \cdots S_{k-1}$ and $\bigodot_{i=0}^{k-1} S_i$ denote the concatenation of $k \in \mathbb{Z}_{\ge 0}$ strings $S_0, \ldots, S_{k-1}$. Moreover, $S^k = \bigodot_{i=0}^{k-1} S$ denotes the concatenation of k copies of the same string S.

We use $\overline{S}$ to denote the reverse of S, that is, $\overline{S} = S[|S|-1]\cdots S[0]$. An integer $p \in (0\,..\,|S|]$ is a period of a string S if $S[i] = S[i+p]$ holds for every $i \in [0\,..\,|S|-p)$; equivalently, p is a period of S if and only if the length-$(|S|-p)$ prefix and suffix of S match, that is, $S[0\,..\,|S|-p) = S[p\,..\,|S|)$.

[3] For $a, b \in \mathbb{R}$, denote $[a\,..\,b] = \{k \in \mathbb{Z} : a \le k \le b\}$, $[a\,..\,b) = \{k \in \mathbb{Z} : a \le k < b\}$, and $(a\,..\,b] = \{k \in \mathbb{Z} : a < k \le b\}$.

2.2 Straight-line Grammars

For a fixed context-free grammar $\mathcal{G}$, we denote by Σ and $\mathcal{N}$ the sets of terminals and non-terminals, respectively. The set of *symbols* is $\mathcal{S} := \Sigma \cup \mathcal{N}$. We say that $\mathcal{G}$ is a *straight-line grammar* (SLG) if:

- each non-terminal $A \in \mathcal{N}$ has a unique production $A \to \mathsf{rhs}(A)$, whose right-hand side is a non-empty sequence of symbols, i.e., $\mathsf{rhs}(A) \in \mathcal{S}^+$, and
- the set of symbols $\mathcal{S}$ admits a partial order $\prec$ such that $B \prec A$ if B occurs in $\mathsf{rhs}(A)$.

A straight-line grammar $\mathcal{G}$ is a *run-length straight-line program* (RLSLP) if each production $A \to \mathsf{rhs}(A)$ is of one of the following two types:

Pair: $\mathsf{rhs}(A) = BC$ for some symbols $B, C \in \mathcal{S}$ such that $B \neq C$;
Power: $\mathsf{rhs}(A) = B^m$ for a symbol $B \in \mathcal{S}$ and an integer $m \geq 2$.

Every straight-line grammar $\mathcal{G}$ yields an *expansion* function $\mathsf{exp} : \mathcal{S}^+ \to \Sigma^+$ assigning to every string $A \in \mathcal{S}^+$ the unique terminal-symbol string $\mathsf{exp}(A) \in \Sigma^+$ derivable from A. The function exp also admits a concise recursive definition:

$$\mathsf{exp}(A) = \begin{cases} A & \text{if } A \in \Sigma, \\ \mathsf{exp}(\mathsf{rhs}(A)) & \text{if } A \in \mathcal{N}, \\ \bigodot_{i=0}^{a-1} \mathsf{exp}(A[\,i\,]) & \text{if } A \in \mathcal{S}^a \text{ for } a > 1. \end{cases}$$

The expansion of the starting symbol of $\mathcal{G}$ is the string *represented* by $\mathcal{G}$.

2.3 Restricted Recompression

Given a string $T \in \Sigma^*$, both recompression [19, 20] and restricted recompression [2, 21] construct a sequence of strings $(T_k)_{k=0}^{\infty}$ over an infinite alphabet $\mathcal{A}$ defined as the least fixed point of the following equation:

$$\mathcal{A} = \Sigma \cup (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{A} \times \mathbb{Z}_{\geq 2}).$$

In other words, $\mathcal{A} = \bigcup_{k=0}^{\infty} \mathcal{A}_k$, where $\mathcal{A}_0 = \Sigma$ and $\mathcal{A}_k = \mathcal{A}_{k-1} \cup (\mathcal{A}_{k-1} \times \mathcal{A}_{k-1}) \cup (\mathcal{A}_{k-1} \times \mathbb{Z}_{\geq 2})$ for $k \in \mathbb{Z}_{>0}$. Symbols in $\mathcal{A} \setminus \Sigma$ are non-terminals with $\mathsf{rhs}((B, C)) = BC$ for $(B, C) \in \mathcal{A} \times \mathcal{A}$ and $\mathsf{rhs}((B, m)) = B^m$ for $(B, m) \in \mathcal{A} \times \mathbb{Z}_{\geq 2}$. Intuitively, $\mathcal{A}$ can be interpreted as a universal RLSLP: for every RLSLP with symbols $\mathcal{S}$ and terminals $\Sigma \subseteq \mathcal{S}$, there is a unique homomorphism $f : \mathcal{S}^* \to \mathcal{A}^*$ such that $f(A) = A$ if $A \in \Sigma$, $\mathsf{rhs}(f(A)) = f(\mathsf{rhs}(A))$ if $A \in \mathcal{S} \setminus \Sigma$, and $f(A) = \bigodot_{i=0}^{|A|-1} f(A[\,i\,])$ for every $A \in \mathcal{S}^*$. As a result, $\mathcal{A}$ provides a convenient formalism for reasoning about procedures generating RLSLPs.

Restricted recompression uses the following transformations of $\mathcal{A}^*$ to $\mathcal{A}^*$.

Definition 4 (Restricted run-length encoding [2, 21]) Given $T \in \mathcal{A}^*$ and $\mathcal{B} \subseteq \mathcal{A}$, we define $\mathsf{rle}_{\mathcal{B}}(T) \in \mathcal{A}^*$ to be the string obtained as follows by decomposing T into blocks and collapsing these blocks:

1. For $i \in [\,0\,..\,|T|-1\,)$, place a *block boundary* between $T[\,i\,]$ and $T[\,i+1\,]$ unless $T[\,i\,] = T[\,i+1\,] \in \mathcal{B}$.
2. Replace each block $T[\,i\,..\,i+m\,) = A^m$ of length $m \geq 2$ with a symbol (A, m).

Remark 5 By construction, $\mathsf{exp}(\mathsf{rle}_{\mathcal{B}}(T)) = \mathsf{exp}(T)$ holds for every $T \in \mathcal{A}^*$ and $\mathcal{B} \subseteq \mathcal{A}$.

Definition 6 (Restricted pair compression [2, 21]) Given $T \in \mathcal{A}^*$ and disjoint sets $\mathcal{L}, \mathcal{R} \subseteq \mathcal{A}$, we define $\mathsf{pc}_{\mathcal{L},\mathcal{R}}(T) \in \mathcal{A}^*$ to be the string obtained as follows by decomposing T into blocks and collapsing these blocks:

1. For $i \in [\,0\,..\,|T|-1\,)$, place a *block boundary* between $T[\,i\,]$ and $T[\,i+1\,]$ unless $T[\,i\,] \in \mathcal{L}$ and $T[\,i+1\,] \in \mathcal{R}$.
2. Replace each block $T[\,i\,..\,i+1\,]$ with a symbol $(T[\,i\,], T[\,i+1\,])$.

Remark 7 By construction, $\mathsf{exp}(\mathsf{pc}_{\mathcal{L},\mathcal{R}}(T)) = \mathsf{exp}(T)$ holds for every $T \in \mathcal{A}^*$ and disjoint $\mathcal{L}, \mathcal{R} \subseteq \mathcal{A}$.

Restricted recompression repeatedly transforms the input text alternating between restricted run-length encoding and restricted pair compression. Unlike in the classic (non-restricted) recompression [19, 20], it is allowed to set $\mathcal{B} \subsetneq \mathcal{A}$ and $\mathcal{L} \cup \mathcal{R} \subsetneq \mathcal{A}$.

Definition 8 (Restricted recompression [2, 21]) For a string $T \in \Sigma^*$, we define $T_0 = T$ and $T_k = \mathsf{shrink}_k(T_{k-1})$ for every $k \in \mathbb{Z}_{>0}$, where

$$\mathsf{shrink}_k = \begin{cases} \mathsf{rle}_{\mathcal{B}_k} & \text{for some fixed set } \mathcal{B}_k \subseteq \mathcal{A} \text{ if } k \text{ is odd,} \\ \mathsf{pc}_{\mathcal{L}_k,\mathcal{R}_k} & \text{for some fixed disjoint sets } \mathcal{L}_k, \mathcal{R}_k \subseteq \mathcal{A} \text{ if } k \text{ is even.} \end{cases}$$

If $r = \min\{k \in \mathbb{Z}_{\geq 0} : |T_k| = 1\}$ exists,[4] we define the underlying *r-round restricted recompression RLSLP* $\mathcal{G}$ to be an RLSLP generating T with symbols $\mathcal{S} = \bigcup_{k=0}^{r} \mathcal{S}_k$, where $\mathcal{S}_k := \{T_k[\,j\,] : j \in [\,0\,..\,|T_k|\,)\}$, and starting symbol $T_r[\,0\,]$.

Our query algorithm assumes that the text T is represented as an r-round restricted recompression RLSLP with every $A \in \mathcal{N}$ storing its right-hand side $\mathsf{rhs}(A)$, expansion length $\|A\| := |\mathsf{exp}(A)|$, and *level*

$$\mathsf{lvl}(A) := \min\{k \in \mathbb{Z}_{\geq 0} : A \in \mathcal{S}_k\}.$$

Example 9 (Restricted recompression) Let us construct a restricted recompression RLSLP for the string $T = \mathtt{bbabbabbabbababababaaaabbab}$, as shown in Fig. 1. After run-length compression, we get $T_1 = B\mathtt{a}B\mathtt{a}B\mathtt{a}B\mathtt{ababab}AB\mathtt{ab}$ and rules $A \to \mathtt{a}^4, B \to \mathtt{b}^2$. Next, we perform pair compression for $\mathcal{L} = \{\mathtt{a}, A\}$ and $\mathcal{R} = \{\mathtt{b}, B\}$. This results in $T_2 = BCCCDDDED$ and rules $C \to \mathtt{a}B, D \to$

[4] Constructions in [2, 19, 21] provide several strategies of picking $\mathcal{B}_k$, $\mathcal{L}_k$, and $\mathcal{R}_k$ based on a given text $T \in \Sigma^n$ so that $|T_r| = 1$ holds for some $r = \mathcal{O}(\log n)$. There are also randomized constructions [2, 7], where $\mathcal{B}_k$, $\mathcal{L}_k$, and $\mathcal{R}_k$ are random variables, and, for every $T \in \Sigma^n$, with high probability, $|T_r| = 1$ holds for some $r = \mathcal{O}(\log n)$.

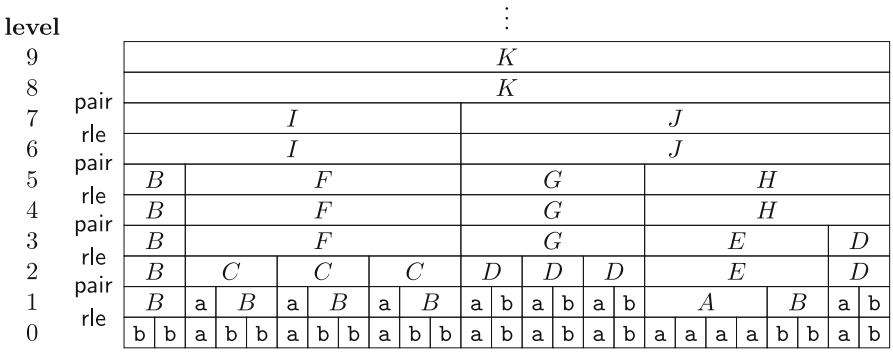


Fig. 1 Construction of the recompression RLSLP from Example 9

ab, $E \to AB$. The next round of run-length compression creates $T_3 = BFGED$ and rules $F \to C^3$, $G \to D^3$. Since there are no runs of power greater than 1, we get $T_4 = T_3$. For the next round of pair compression, we pick $\mathcal{L} = \{B, G, E\}$ and $\mathcal{R} = \{D\}$. Note that we did not assign F at all. Since we perform restricted recompression, this is allowed. This results in $T_5 = BFGH = T_6$ with $H \to ED$. During the following three rounds, we obtain $|T_8| = 1$ and $T_9 = T_{10} = T_{11} = \ldots$.

We obtained the grammar $A \to a^4$, $B \to b^2$, $C \to aB$, $D \to ab$, $E \to AB$, $F \to C^3$, $G \to D^3$, $H \to ED$, $I \to BF$, $J \to GH$, $K \to IJ$.

2.4 Parse Tree

The *parse tree* $\mathcal{T}(A)$ of a symbol $A \in \mathcal{S}$ in a straight-line grammar is a rooted ordered tree with each node v associated to a symbol $v.\mathsf{symb} \in \mathcal{S}$. The root of $\mathcal{T}(A)$ is a node ρ with $\rho.\mathsf{symb} = A$. If $A \in \Sigma$, then ρ has no children. If $A \in \mathcal{N}$ and $\mathsf{rhs}(A) = A_0 \cdots A_{a-1}$, then ρ has a children, and the subtree rooted at the ith child is (a copy of) $\mathcal{T}(A_i)$. The parse tree $\mathcal{T}$ of a straight-line grammar $\mathcal{G}$ is the parse tree of the starting symbol of $\mathcal{G}$ (whose expansion is the text T represented by $\mathcal{G}$).

Each node v of $\mathcal{T}$ is also associated with a position $v.\mathsf{pos} \in [\,0\,..\,|T|\,)$, which we interpret as the index of the leftmost leaf in the subtree of $\mathcal{T}$ rooted at v. Moreover, we denote by $\mathsf{exp}(v) := T[\,v.\mathsf{pos}\,..\,v.\mathsf{pos}+\|v.\mathsf{symb}\|\,)$ the fragment of T corresponding to v. For the root ρ, we define $\rho.\mathsf{pos} = 0$ so that $\mathsf{exp}(\rho) = T[\,0\,..\,|T|\,)$ is the whole T. Moreover, if $\mathsf{rhs}(v.\mathsf{symb}) = A_0 \cdots A_{a-1}$ and $v_0, \ldots, v_{a-1}$ are the children of v, then $v_i.\mathsf{pos} = v.\mathsf{pos} + \sum_{j=0}^{i-1} \|A_j\|$ for $i \in [\,0\,..\,a\,)$. This way, $\mathsf{exp}(v)$ is the concatenation of fragments $\mathsf{exp}(v_0), \ldots, \mathsf{exp}(v_{a-1})$. A straightforward inductive argument shows that $\mathsf{exp}(v)$ is an occurrence of $\mathsf{exp}(v.\mathsf{symb})$ in $\mathcal{T}$.

Although we typically cannot afford to store the parse tree $\mathcal{T}$ explicitly, our representation of the restricted recompression RLSLP allows for efficiently traversing the parse tree $\mathcal{T}$ using an abstraction of *pointers*. We implement a pointer to a node $v \in \mathcal{T}$ as a tuple $(v.\mathsf{pos}, v.\mathsf{symb}, v.\mathtt{parent})$, where $v.\mathtt{parent}$ is (a pointer to) the parent v (or **nil** if v is the root of $\mathcal{T}$). The symbol $v.\mathsf{symb}$ and the parent $v.\mathtt{parent}$ are stored as constant references (allowing many references to the same immutable object

in the computer memory). This way, one can think of a pointer as a stack containing $\mu.\mathsf{pos}$ and $\mu.\mathsf{symb}$ for every ancestor μ of ν starting from the root ρ at the bottom of the stack to the node ν itself at the top of the stack. The stack implementation is like in functional programming languages so that multiple pointers can share a common prefix of the stack.

Given (a pointer to) a node ν, we can in constant time retrieve (a pointer to) the parent $\nu.\mathtt{parent}$. Given additionally an index i, we can also construct (a pointer to) the ith child of ν, denoted $\nu.\mathtt{child}(i)$; we assume that $\nu.\mathtt{child}(i) = \mathbf{nil}$ for out-of-bounds indices i, that is, for $i \notin [0\,..\,|\mathsf{rhs}(\nu.\mathsf{symb})|)$. Additionally, given any position $j \in [\,\nu.\mathsf{pos}\,..\,\nu.\mathsf{pos} + \|\nu.\mathsf{symb}\|\,)$, we can compute an index $i := \nu.\mathtt{index}(j)$ of the unique child ν_i of ν such that $j \in [\,\nu_i.\mathsf{pos}\,..\,\nu_i.\mathsf{pos} + \|\nu_i.\mathsf{symb}\|\,)$. If $\nu.\mathsf{symb}$ is a run, then $i = \lfloor (j - \nu.\mathsf{pos})/\|\nu.\mathtt{child}(0).\mathsf{symb}\| \rfloor$. If $\nu.\mathsf{symb}$ is not a run, then it has exactly two children. If $\nu.\mathsf{pos} + \|\nu.\mathtt{child}(0).\mathsf{symb}\| > j$, then $i = 0$, otherwise $i = 1$. Using this functionality, we can descend to the desired leaf of $\mathcal{T}$:

Observation 10 *Given any position $j \in [\,0\,..\,|T|\,)$, a pointer to the j-th leaf of $\mathcal{T}$, that is, a node $\mathtt{leaf}(j) \in \mathcal{T}$ such that $\mathsf{exp}(\mathtt{leaf}(j)) = T[\,j\,]$, can be retrieved in $\mathcal{O}(r)$ time.*

Proof We define a more general recursive $\nu.\mathtt{leaf}(j)$ function that computes $\mathtt{leaf}(j)$ given (a pointer to) a node ν such that $j \in [\,\nu.\mathsf{pos}\,..\,\nu.\mathsf{pos} + \|\nu.\mathsf{symb}\|\,)$. If ν is already a leaf (i.e., $\|\nu.\mathsf{symb}\| = 1$), then $\nu.\mathtt{leaf}(j) = \nu$. Otherwise, $\nu.\mathtt{leaf}(j) = \nu.\mathtt{child}(\nu.\mathtt{index}(j)).\mathtt{leaf}(j)$. Finally, we observe that $\mathtt{leaf}(j)$ can be computed as $\rho.\mathtt{leaf}(j)$, where ρ is the root of $\mathcal{T}$.

The time complexity is proportional to the length of the path from ρ to $\mathtt{leaf}(j)$, which is $\mathcal{O}(r)$. □

Remark 11 Every pointer can be stored in a constant number of machine words. Every pointer operation allocates a constant amount of new space. The query algorithm of Observation 10 starts at the root and calls $.\mathtt{child}()$ up to r times. Therefore, this algorithm uses $\mathcal{O}(r)$ space.

If $\mu = \nu.\mathtt{parent}$ and $\mu_0, \ldots, \mu_{a-1}$ are the children of μ, then we can retrieve the index i such that $\nu = \mu_i$ as $i := \nu.\mathtt{parent}.\mathtt{index}(\nu.\mathsf{pos})$. For $d \in [\,-i\,..\,a-i\,)$, we refer to μ_{i+d} as the dth sibling of ν and, for convenience, write $\nu.\mathtt{sibling}(d) := \nu.\mathtt{parent}.\mathtt{child}(d + \nu.\mathtt{parent}.\mathtt{index}(\nu.\mathsf{pos}))$. We assume that $\nu.\mathtt{sibling}(d) = \mathbf{nil}$ if $d \notin [\,-i\,..\,a-i\,)$ or $\mu = \nu.\mathtt{parent} = \mathbf{nil}$.

Example 12 (Parse trees) The parse tree immediately follows from the grammar we constructed. The main difference comes from the pointer representation. In Fig. 2, we construct the parse tree for our grammar from the running example.

2.5 Uncompacted Parse Tree

We also observe that each symbol in each of the strings $(T_k)_{k=0}^{\infty}$ can be associated with a node in the parse tree $\mathcal{T}$. This mapping is not injective, though: if a symbol of T_k forms

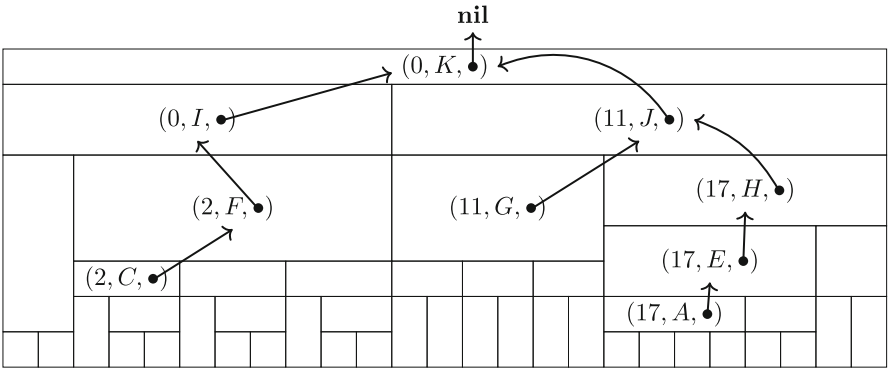


Fig. 2 The (compacted) parse tree created from the grammar in Fig. 1. Only partially labelled

a length-one block with respect to shrink_{k+1}, it remains in T_{k+1} represented by the same node of the parse tree $\mathcal{T}$. As in [7], this can be alleviated using an *uncompacted parse tree* $\overline{\mathcal{T}}$, where the edges of $\mathcal{T}$ are subdivided so that a node $v \in \mathcal{T}$ gets a separate copy for each level k such that T_k has a symbol associated with v. As a result, there is a bijection between the characters $T_k[\,i\,]$, where $k \in \mathbb{Z}_{\geq 0}$ and $i \in [\,0\,..\,|T_k|\,)$, and the nodes $\overline{v}$ of $\overline{\mathcal{T}}$.[5]

For each node $\overline{v} \in \overline{\mathcal{T}}$, we denote $\overline{v}.\mathsf{level}$ and $\overline{v}.\mathsf{symb}$ to be the level of $\overline{v}$ and the symbol represented by $\overline{v}$, respectively; in other words, $\overline{v}.\mathsf{level} = k$ and $\overline{v}.\mathsf{symb} = A$ if and only if $\overline{v}$ represents $T_k[\,i\,] = A$ for some $i \in [\,0\,..\,|T_k|\,)$. Note that $\overline{v}.\mathsf{level} \geq \mathsf{lvl}(\overline{v}.\mathsf{symb})$, but the equality does not necessarily hold.

Fact 13 *A node $\overline{v} \in \overline{\mathcal{T}}$ has exactly one child if and only if $\overline{v}.\mathsf{level} > \mathsf{lvl}(\overline{v}.\mathsf{symb})$.*

Proof Suppose that $\overline{v}$ represents $T_k[\,i\,] = A$ for some $i \in [\,0\,..\,|T_k|\,)$, where $\overline{v}.\mathsf{level} = k$ and $\overline{v}.\mathsf{symb} = A$. Moreover, let $\ell = \mathsf{lvl}(A)$. If $k = \ell = 0$, then, trivially, $\overline{v}$ has no children. If $k = \ell > 0$, then, by definition of ℓ, the symbol A cannot occur in T_{k-1}, so $T_k[\,i\,]$ must have been obtained by collapsing a block of T_{k-1} (with respect to shrink_k) that consists of $|\mathsf{rhs}(A)| \geq 2$ symbols, and thus $\overline{v}$ has at least two children.

Finally, for a proof by contradiction, suppose that $k > \ell$ yet $\overline{v}$ has at least two children (it cannot have a single child since $k > 0$ and every block with respect to shrink_k is non-empty). This means that $T_k[\,i\,]$ must have been obtained by collapsing a block $T_{k-1}[\,i'\,..\,j'\,)$ (with respect to shrink_k) of length $j' - i' \geq 2$. Further expansions of this block yield a fragment $T_\ell[\,i''\,..\,j''\,)$ of length $j'' - i'' \geq j' - i' \geq 2$ whose expansion matches $\mathsf{exp}(T_k[\,i\,]) = \mathsf{exp}(A)$. At the same time, by definition of ℓ, the symbol A occurs as one of the characters (length-1 fragments) of T_ℓ. This contradicts [2, Lemma 4.4], which states that two fragments of T_ℓ match if and only if they have matching expansions. □

We use $\overline{v}.\texttt{parent}$ and $\overline{v}.\texttt{child}(i)$ to denote the parent and the ith child of $\overline{v}$, respectively (with the latter defined as **nil** for out-of-bounds arguments i). Observe that

[5] To support arbitrarily large $k \in \mathbb{Z}_{\geq 0}$, the root of $\mathcal{T}$ corresponds to an infinite path in $\overline{\mathcal{T}}$, with one node per level $k \geq r$.

every node $\overline{v} \in \overline{\mathcal{T}}$ satisfies $\overline{v}.\texttt{parent}.\mathsf{level} = \overline{v}.\mathsf{level} + 1$ and, if $\overline{v}.\texttt{child}(i) \neq \textbf{nil}$, then also $\overline{v}.\texttt{child}(i).\mathsf{level} = \overline{v}.\mathsf{level} - 1$. Fact 13 lets us easily implement the parent and child operations based on the analogous operations on $\mathcal{T}$:

Observation 14 *Given a (pointer to) a node $\overline{v} \in \overline{\mathcal{T}}$, one can in constant time compute (pointers to) $\overline{v}.\texttt{parent}$ as well as $\overline{v}.\texttt{child}(i)$ for any index $i \in \mathbb{Z}$.*

Proof Suppose that $\overline{v}$ is represented as (v, k), where v is the corresponding node of $\mathcal{T}$ and $k = \overline{v}.\mathsf{level}$. If $v.\texttt{parent} \neq \textbf{nil}$ (i.e., v is not the root of $\mathcal{T}$) and $\mathsf{lvl}(v.\texttt{parent}.\mathsf{symb}) = k + 1$, then, by Fact 13, we have $\overline{v}.\texttt{parent} = (v.\texttt{parent}, k + 1)$; otherwise, $\overline{v}.\texttt{parent} = (v, k + 1)$.

Furthermore, $\overline{v}.\texttt{child}(0) = (v, k - 1)$ if $k > \mathsf{lvl}(v.\mathsf{symb})$, and $\overline{v}.\texttt{child}(i) = (v.\texttt{child}(i), k - 1)$ if $k = \mathsf{lvl}(v.\mathsf{symb})$ and $v.\texttt{child}(i) \neq \textbf{nil}$. In the remaining cases, $\overline{v}.\texttt{child}(i) = \textbf{nil}$. □

If $\overline{v}$ represents $T_k[\,i\,]$, then we use $\overline{v}.\texttt{next}()$ and $\overline{v}.\texttt{prev}()$ to denote nodes representing $T_k[\,i+1\,]$ and $T_k[\,i-1\,]$, respectively, with $\overline{v}.\texttt{next}() = \textbf{nil}$ if $i = |T_k| - 1$ and $\overline{v}.\texttt{prev}() = \textbf{nil}$ if $i = 0$. A naive implementation of the operation computing $\overline{\mu} = \overline{v}.\texttt{next}()$ traverses the path from $\overline{v}$ to $\overline{\mu}$ in the uncompacted parse tree $\overline{\mathcal{T}}$, and symmetrically for $\overline{v} = \overline{\mu}.\texttt{prev}()$. As shown in [7], a more sophisticated implementation of pointers (which requires additional auxiliary data structures and an assumption that the machine word consists of $\Omega(r)$ bits) allows for a constant-time implementation. In this work, we use a convenient middle-ground implementation based on the parse tree $\mathcal{T}$:

Lemma 15 *The $\cdot.\texttt{next}()$ operation on uncompacted parse tree nodes can be implemented so that, for every node $\overline{v}_0 \in \overline{\mathcal{T}}$, every sequence of s valid operations $(\overline{v}_i := \overline{v}_{i-1}.\mathsf{op}_i)_{i=1}^{s}$ takes $\mathcal{O}(r+s)$ time provided that, for each $i \in [\,1\,..\,s\,]$, the operation $\overline{v}_i := \overline{v}_{i-1}.\mathsf{op}_i$ is of the form $\overline{v}_i := \overline{v}_{i-1}.\texttt{next}()$, $\overline{v}_i := \overline{v}_{i-1}.\texttt{parent}$, or $\overline{v}_i := \overline{v}_{i-1}.\texttt{child}(j)$ for some $j \in \mathbb{Z}$. The $\cdot.\texttt{prev}()$ operation satisfies the same condition.*[6]

Proof Let us first describe the implementation of $\overline{v}.\texttt{next}()$ provided that $\overline{v} \in \overline{\mathcal{T}}$ is represented by (v, k), where v is the corresponding node of $\mathcal{T}$ and $k = \overline{v}.\mathsf{level}$. Suppose that $\overline{v}.\texttt{next}() = \overline{\mu}$, where $\overline{\mu} \in \overline{\mathcal{T}}$ is analogously represented by (μ, k). Our algorithm traverses the path from v to μ in $\mathcal{T}$. For this, we first traverse the path from v towards the root of $\mathcal{T}$ (using the $\cdot.\texttt{parent}$ operation) until we encounter a node that has a right sibling. If this does not happen before we reach the root of $\mathcal{T}$, then $\overline{v}.\texttt{next}() = \textbf{nil}$. Otherwise, we proceed to the right sibling (using the $\cdot.\texttt{sibling}(1)$ operation) and then descend $\mathcal{T}$ (using the $\cdot.\texttt{child}(0)$ operation) until we reach μ, i.e., the first node that we encounter that satisfies $\mathsf{lvl}(\mu.\mathsf{symb}) \leq k$. Our implementation of pointers to $\mathcal{T}$ guarantees that the running time of this procedure is proportional to the length of the path from v to μ in $\mathcal{T}$.

[6] If $\cdot.\texttt{prev}()$ and $\cdot.\texttt{next}()$ are both used in the sequence of s operations, we do not guarantee the $\mathcal{O}(r+s)$ running time.

Consider a sequence $(\overline{v}_i := \overline{v}_{i-1}.\mathsf{op}_i)_{i=1}^{s}$ of s calls, and, for $i \in [\![0\,..\,s]\!]$, denote by v_i the nodes of $\mathcal{T}$ corresponding to $\overline{v}_i$. Observe that the paths from v_i to v_{i+1} form subsequent fragments of the Euler tour of the subtree of $\mathcal{T}$ consisting of all the nodes $v_0, \ldots, v_s$ and their ancestors. Since each internal node in $\mathcal{T}$ has at least two children, the subtree size is $\mathcal{O}(s + r)$: besides the $s + 1$ nodes $v_0, \ldots, v_s$ and their at most s lowest common ancestors, it only contains $\mathcal{O}(r)$ ancestors of v_0 and v_s.

The implementation and analysis of the $\cdot.\texttt{prev}()$ operation is symmetric. □

Example 16 (Uncompacted parse tree) The uncompacted parse tree is derived from the compacted parse tree. It is isomorphic to the blocks created during the construction of the grammar.

Observe in Fig. 3 that $\alpha.\texttt{parent} = \beta$, and $\beta.\texttt{parent} = \gamma$, and $\gamma.\texttt{parent} = \nu$. Further, $\beta.\texttt{child}(0) = \alpha$ and $\gamma.\texttt{child}(0) = \beta$, and $\nu.\texttt{child}(1) = \gamma$. Note that $\beta.\texttt{child}(1) = \textbf{nil}$.

The symbol operator returns the symbol in the grammar for some node in the parse tree, like so $\alpha.\mathsf{symb} = a$.

The level of terminals is 0, thus $\mathsf{lvl}(a) = 0 = \mathsf{lvl}(\alpha)$. We can see that $\mathsf{lvl}(\beta) = 1$, which is strictly greater than $\mathsf{lvl}(\beta.\mathsf{symb})$.

3 Popped Sequences and LCE Queries

Efficient implementation of LCE queries on recompression RLSLPs relies on a notion of *popped sequences* introduced by I [5]. In this section, we provide a slightly simpler construction that is not tailored to any specific version of recompression. We then prove basic properties of popped sequences and, as a simple application, recover the LCE functionality of [6] without relying on the properties of the specific recompression RLSLP constructed in [6, Section 6].

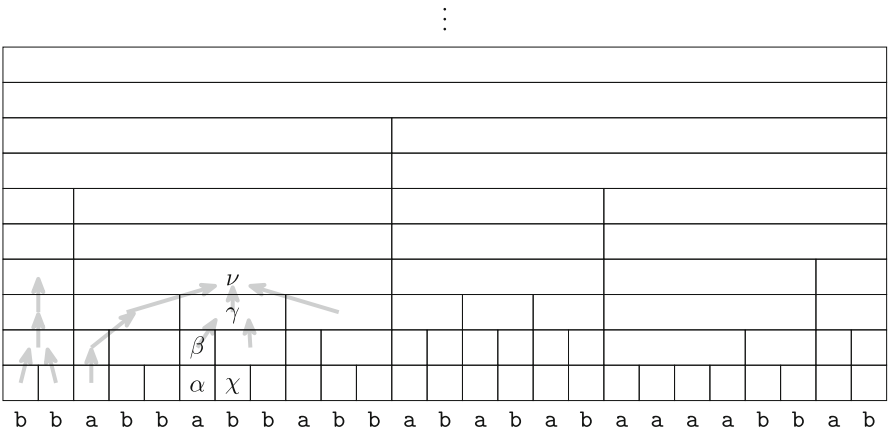


Fig. 3 The uncompacted parse tree created from the grammar in Fig. 1. Arrows point from a node to its parent. Only partially labelled

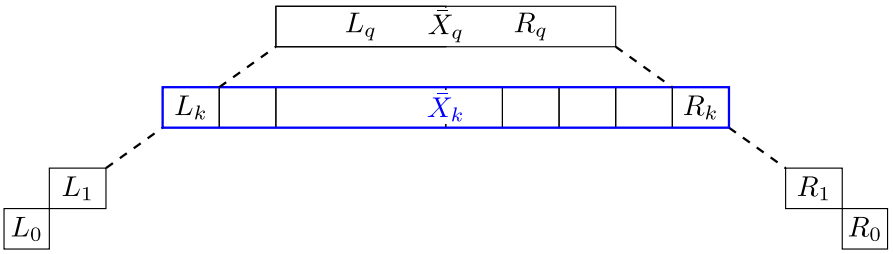


Fig. 4 The popped sequence is build from the blocks L_0 through R_0. In every level k, the string $\bar{X}_k$ spans from L_k to R_k (inclusive)

3.1 Popped Sequences

Throughout this section, we fix a text $T \in \Sigma^n$ and an arbitrary r-round recompression RLSLP $\mathcal{G}$ generating T; note that $\mathcal{G}$ is uniquely determined by T and the sets $\mathcal{B}_k$ (for odd $k \in \mathbb{Z}_{>0}$) as well as $\mathcal{L}_k$ and $\mathcal{R}_k$ (for even $k \in \mathbb{Z}_{>0}$) of Definition 8.

Definition 17 (Popped Sequence; see Fig. 4) Given a string $X \in \Sigma^*$, we define strings $L_k, R_k, \bar{X}_k \in \mathcal{A}^*$ for $k \in \mathbb{Z}_{\geq 0}$ as follows. We set $\bar{X}_0 = X$. For $k \geq 0$, we define L_k, R_k, and $\bar{X}_{k+1}$ depending on the blocks of $\mathsf{shrink}_{k+1}(\bar{X}_k)$:

- L_k is of the leftmost block of $\bar{X}_k$ (with respect to shrink_{k+1}) unless $\bar{X}_k$ is partitioned into zero blocks (i.e., $\bar{X}_k = \varepsilon$) or the leftmost block of $\bar{X}_k$ consists of two distinct symbols. In these two special cases, $L_k = \varepsilon$.
- R_k is of the rightmost block of $\bar{X}_k$ (with respect to shrink_{k+1}) unless $\bar{X}_k$ is partitioned into at most one block or the rightmost block of $\bar{X}_k$ consist of two distinct symbols. In these two special cases, $R_k = \varepsilon$.
- $\bar{X}_{k+1} = \mathsf{shrink}_{k+1}(\bar{X}'_k)$, where $\bar{X}'_k \in \mathcal{A}^*$ is the substring of $\bar{X}_k$ such $\bar{X}_k = L_k \cdot \bar{X}'_k \cdot R_k$.

The *popped sequence* of X is $\mathsf{PSeq}(X) := L_0 \cdots L_q \cdot R_q \cdots R_0$ for any $q \in \mathbb{Z}_{\geq 0}$ such that $\bar{X}_{q+1} = \varepsilon$.

In other words, to construct the sequence $(\bar{X}_k)_{k=0}^{\infty}$, we proceed as in Definition 8, but we remove the leftmost and the rightmost block of $\bar{X}_k$ before we collapse the remaining blocks to obtain $\bar{X}_{k+1}$, unless here are no blocks left to be removed or we would remove a block of two distinct symbols (we keep such a block and the symbol it collapses into will be removed at the next level, either on its own or as a part of a longer run). The removed blocks are concatenated (in the appropriate order) to form $\mathsf{PSeq}(X)$. The following observation captures the immediate consequences of the Definition 17.

Observation 18 *For every $k \in \mathbb{Z}_{\geq 0}$, we have $|\mathsf{shrink}_{k+1}(L_k)| \leq 1$, $|\mathsf{shrink}_{k+1}(R_k)| \leq 1$, $|\mathsf{rle}(L_k)| \leq 1$ and $|\mathsf{rle}(R_k)| \leq 1$, as well as $X = \mathsf{exp}(L_0 \cdots L_{k-1} \cdot \bar{X}_k \cdot R_{k-1} \cdots R_0)$.*

A crucial property of the popped sequence is that every occurrence of X in a text T induces the occurrences of each non-empty string $\bar{X}_k$ in the string T_k, and thus an "occurrence" of the entire popped sequence in the parse tree of T.

Lemma 19 *If $X \in \Sigma^+$ occurs in a text T at position i and $\bar{X}_k \neq \varepsilon$, then $\bar{X}_k$ occurs in T_k at a position i_k such that $|\mathsf{exp}(T_k[\,0\,..\,i_k\,))| = i + |\mathsf{exp}(L_0 \cdots L_{k-1})|$.*

Proof We proceed by induction on k. The base case is trivial since $T_0 = T$ and $\bar{X}_0 = X$. For $k \in \mathbb{Z}_{>0}$, the inductive hypothesis yields an occurrence of $\bar{X}_{k-1}$ starting at position i_{k-1}. By Definitions 4 and 6, shrink_k places block boundaries based on the identities of the two surrounding symbols. In particular, the block boundaries strictly inside $\bar{X}_{k-1}$ are placed in the same way regardless of whether $\bar{X}_{k-1}$ is processed as a standalone string or as a fragment of T_{k-1}. Moreover, if the leftmost block of $\bar{X}_{k-1}$ consists of two distinct symbols, then there is a block boundary at the beginning of every occurrence of $\bar{X}_{k-1}$ in T_{k-1} — this is because shrink_{k+1} cannot create a block of length at least three consisting of at least two distinct symbols. Symmetrically, if the rightmost block of $\bar{X}_{k-1}$ consists of two distinct symbols, then there is a block boundary at the end of every occurrence of $\bar{X}_{k-1}$ in T_{k-1}. In all cases, the blocks collapsed to $\bar{X}_k$ are processed in the same way and induce an occurrence of $\bar{X}_k$ in T_k. The starting position i_k of this occurrence satisfies $|\mathsf{exp}(T_k[\,0\,..\,i_k\,))| = |\mathsf{exp}(T_{k-1}[\,0\,..\,i_{k-1}\,) \cdot L_{k-1})| = |\mathsf{exp}(T_{k-1}[\,0\,..\,i_{k-1}\,))| + |\mathsf{exp}(L_{k-1})|$, which is $i + |\mathsf{exp}(L_0 \cdots L_{k-2})| + |\mathsf{exp}(L_{k-1})| = i + |\mathsf{exp}(L_0 \cdots L_{k-1})|$ by the inductive hypothesis. □

Next, we address the task of efficiently computing a popped sequence.

Lemma 20 *Let $\mathcal{G}$ be an r-round restricted recompression RLSLP representing a text T. Given a fragment $X = T[\,i\,..\,j\,]$, the run-length encoding of the popped sequence $\mathsf{PSeq}(X)$ can be computed in $\mathcal{O}(r)$ time, along with a decomposition $\mathsf{PSeq}(X) = L_0 \cdots L_q \cdot R_q \cdots R_0$ for $q = \max\{k \in \mathbb{Z}_{\geq 0} : \bar{X}_k \neq \varepsilon\}$.*

Proof For each round k such that $\bar{X}_k \neq \varepsilon$, our algorithm assumes to be given (pointers to) nodes $\overline{v}_k, \overline{\mu}_k \in \overline{\mathcal{T}}$ representing, respectively, the leftmost and the rightmost symbol within the occurrence of $\bar{X}_k$ in T_k stipulated by Lemma 19. For $k = 0$, the nodes $\overline{v}_0$ and $\overline{\mu}_0$ correspond to $T[\,i\,]$ and $T[\,j\,]$, respectively, and they can be obtained in $\mathcal{O}(r)$ time by Observation 10.

For subsequent levels $k \in \mathbb{Z}_{\geq 0}$, the algorithm first considers the parents $\overline{v}'_k := \overline{v}_k.\mathtt{parent}$ and $\overline{\mu}'_k := \overline{\mu}_k.\mathtt{parent}$. The nodes $\overline{v}'_k$ and $\overline{\mu}'_k$ correspond to, respectively, the leftmost and the rightmost block of $\bar{X}_k$ with respect to shrink_{k+1}, possibly including some extra symbols to the left or to the right of $\bar{X}_k$, respectively. In particular, since we already assumed that $\bar{X}_k \neq \varepsilon$, then $L_k = \varepsilon$ if and only if $\overline{v}'_k$ has two children with distinct symbols, $\overline{v}_k$ is the left child of $\overline{v}'_k$, and $\overline{v}_k \neq \overline{\mu}_k$ (if $\overline{v}_k = \overline{\mu}_k$, then the only block of $\bar{X}_k$ consists of a single symbol). Similarly, $R_k = \varepsilon$ if (but not only if) $\overline{\mu}_k$ has two children with distinct symbols and $\overline{\mu}_k$ is the right child of $\overline{\mu}'_k$. The other possibility for $R_k = \varepsilon$ is when $L_k \neq \varepsilon$ and $\overline{v}'_k = \overline{\mu}'_k$ (that is, $\bar{X}_k$ consists of a single block that does not have two distinct symbols). In this special case, $L_k = \bar{X}_k$ is the power of $\overline{v}_k.\mathsf{symb}$ with exponent equal to the number of siblings between $\overline{v}_k$ and $\overline{\mu}_k$ (inclusive).

Except for the special case, we have $\overline{v}_{k+1} = \overline{v}'_k$ if $L_k = \varepsilon$. Otherwise, $\overline{v}_{k+1} = \overline{v}'_k.\mathtt{next}()$ and, by Observation 18, L_k is a power of $\overline{v}_k.\mathsf{symb}$ with exponent equal to the number of right siblings of $\overline{v}_k$ (including $\overline{v}_k$). Symmetrically, $\overline{\mu}_{k+1} = \overline{\mu}'_k$ if $R_k = \varepsilon$. Otherwise, $\overline{\mu}_{k+1} = \overline{\mu}'_k.\mathtt{prev}()$ and, by Observation 18, R_k is a power of $\overline{\mu}_k.\mathsf{symb}$ with exponent equal to the number of left siblings of $\overline{\mu}_k$ (including $\overline{\mu}_k$).

Before we proceed to the next level, we must stop when $\bar{X}_{k+1} = \varepsilon$; this condition holds when $L_k \neq \varepsilon$ and $\overline{v}'_k = \overline{\mu}'_k$ (the special case considered above) as well as when $L_k \neq \varepsilon$, $R_k \neq \varepsilon$, and $\overline{v}'_k.\texttt{next}() = \overline{\mu}'_k$.

The sequence of $\cdot.\texttt{parent}$ and $\cdot.\texttt{next}()$ operations involving nodes $\overline{v}_k$ and $\overline{v}'_k$, as well as the sequence of $\cdot.\texttt{parent}$ and $\cdot.\texttt{prev}()$ operations involving nodes $\overline{\mu}_k$ and $\overline{\mu}'_k$ satisfy the conditions of Lemma 15, so the overall running time is $\mathcal{O}(q + r) = \mathcal{O}(r)$. □

3.2 LCE Queries in $\mathcal{O}(r)$ Time

Recall that an LCE query, given positions i and i' in $T \in \Sigma^n$, returns

$$\mathsf{LCE}_T(i, i') = \max\{d \in [\,0\,..\,n - \max(i, i')\,] : T[\,i\,..\,i + d\,) = T[\,i'\,..\,i' + d\,)\}.$$

As a sample application of our implementation of parse tree nodes and our variant of popped sequences, we show how to obtain $\mathcal{O}(r)$-time LCE queries in our setting. Our recursive procedure answering LCE queries is an easy to implement version of the algorithm originating from [6, Theorem 5.25] and analyzed using popped sequences in the preliminary version of this paper [23]. The algorithm shares many ideas with earlier implementations of Longest Common Extension and Longest Common Prefix queries on locally consistent RLSLPs [5, 8, 15–18].

Proposition 21 *LCE queries on a text T represented using an r-round restricted recompression run-length straight-line program can be answered in $\mathcal{O}(r)$ time.*

Proof We implement a recursive procedure $\texttt{lce}(v, v')$ that, given (pointers to) nodes $v, v' \in \mathcal{T}$, returns $\mathsf{LCE}_T(v.\mathsf{pos}, v'.\mathsf{pos})$. To streamline the presentation, we also assume $\texttt{lce}(v, \textbf{nil}) = \texttt{lce}(\textbf{nil}, v') = 0$.

Our implementation relies on a helper function $v.\texttt{next}()$ that returns the highest node μ such that $\mathsf{exp}(\mu)$ immediately follows $\mathsf{exp}(v)$, that is, $\mu.\mathsf{pos} = v.\mathsf{pos} + \|v.\mathsf{symb}\|$. Formally,

- If $v.\texttt{parent} = \textbf{nil}$, then $v.\texttt{next}() = \textbf{nil}$, i.e., $v.\texttt{next}()$ is undefined if v is the root of $\mathcal{T}$.
- If $v.\texttt{sibling}(1) \neq \textbf{nil}$, then $v.\texttt{next}() = v.\texttt{sibling}(1)$, i.e., $v.\texttt{next}()$ is the right sibling of v if such a sibling exists.
- Otherwise, $v.\texttt{next}() = v.\texttt{parent}.\texttt{next}()$. In this case, v does not have any right sibling, so $v.\mathsf{pos} + \|v.\mathsf{symb}\| = v.\texttt{parent}.\mathsf{pos} + \|v.\texttt{parent}.\mathsf{symb}\|$ and the value $v.\texttt{next}()$ can be computed recursively.

We can now implement $\texttt{lce}(v, v')$ assuming $v \neq \textbf{nil}$ and $v' \neq \textbf{nil}$.

1. If $\|v.\mathsf{symb}\| = \|v'.\mathsf{symb}\| = 1$ and $v.\mathsf{symb} \neq v'.\mathsf{symb}$, then we return 0.
2. If $\|v.\mathsf{symb}\| = \|v'.\mathsf{symb}\| > 1$ and $v.\mathsf{symb} \neq v'.\mathsf{symb}$, then we return $\texttt{lce}(v.\texttt{child}(0), v'.\texttt{child}(0))$.
3. If $\|v.\mathsf{symb}\| > \|v'.\mathsf{symb}\|$, then we return $\texttt{lce}(v.\texttt{child}(0), v')$.
4. If $\|v.\mathsf{symb}\| < \|v'.\mathsf{symb}\|$, then we return $\texttt{lce}(v, v'.\texttt{child}(0))$.

5. If $v.\mathsf{symb} = v'.\mathsf{symb}$ and $v.\texttt{sibling}(1) = \textbf{nil}$ or $v'.\texttt{sibling}(1) = \textbf{nil}$, then we return $\|v.\mathsf{symb}\| + \texttt{lce}(v.\texttt{next}(), v'.\texttt{next}())$.
6. Otherwise, $v.\mathsf{symb} = v'.\mathsf{symb}$ and $v.\texttt{sibling}(1) \neq \textbf{nil} \neq v.\texttt{sibling}(1)$. We pick the largest d such that $v.\texttt{sibling}(d) \neq \textbf{nil} \neq v.\texttt{sibling}(d)$ and return $d \cdot \|v.\mathsf{symb}\| + \texttt{lce}(v.\texttt{sibling}(d), v'.\texttt{sibling}(d))$.

Let us justify the correctness of the algorithm. In case 1, we have $T[\,v.\mathsf{pos}\,] = v.\mathsf{symb} \neq v'.\mathsf{symb} = T[\,v'.\mathsf{pos}\,]$, so the algorithm correctly returns $\mathsf{LCE}_T(v.\mathsf{pos}, v'.\mathsf{pos}) = 0$. In the remaining cases, we make a recursive call, which we denote $\texttt{lce}(\mu, \mu')$. In cases 2–4, we have $\mu.\mathsf{pos} = v.\mathsf{pos}$ and $\mu'.\mathsf{pos} = v'.\mathsf{pos}$, so the value $\mathsf{LCE}_T(v.\mathsf{pos}, v'.\mathsf{pos}) = \mathsf{LCE}_T(\mu.\mathsf{pos}, \mu'.\mathsf{pos})$ is computed correctly. Moreover, whenever $\mu = v.\texttt{child}(0)$ or $\mu' = v'.\texttt{child}(0)$, we have $\|v.\mathsf{symb}\| > 1$ or $\|v'.\mathsf{symb}\| > 1$, respectively, so the leftmost children exist. In the remaining cases, we have $v.\mathsf{symb} = v'.\mathsf{symb}$ and thus $\mathsf{LCE}_T(v.\mathsf{pos}, v'.\mathsf{pos}) = \|v.\mathsf{symb}\| + \mathsf{LCE}_T(v.\mathsf{pos} + \|v.\mathsf{symb}\|, v'.\mathsf{pos} + \|v.\mathsf{symb}\|)$. In case 5, we specifically pick $\mu = v.\texttt{next}()$ and $\mu' = v'.\texttt{next}()$ so that $\mu.\mathsf{pos} = v.\mathsf{pos} + \|v.\mathsf{symb}\|$ and $\mu'.\mathsf{pos} = v'.\mathsf{pos} + \|v'.\mathsf{symb}\|$, and hence the returned value is correct. (This argument works even if $\mu = \textbf{nil}$ or $\mu' = \textbf{nil}$ as long as we assume $\textbf{nil}.\mathsf{pos} = |T|$.) The same is true in case 6 provided that $d = 1$. If $d \geq 2$, on the other hand, then the parents of v and v' have at least three children each, so the underlying symbols must be runs. In this case, all the siblings of v and v' have the same symbol $v.\mathsf{symb}$. Consequently, $\mathsf{LCE}_T(v.\mathsf{pos}, v'.\mathsf{pos}) = d \cdot \|v.\mathsf{symb}\| + \mathsf{LCE}_T(v.\mathsf{pos} + d \cdot \|v.\mathsf{symb}\|, v'.\mathsf{pos} + d \cdot \|v.\mathsf{symb}\|) = d \cdot \|v.\mathsf{symb}\| + \mathsf{LCE}_T(\mu.\mathsf{pos}, \mu'.\mathsf{pos})$, and hence the returned value is correct. Finally, observe that the algorithm terminates because, throughout the recursion, both v and v' move forward with respect to the pre-order traversal of $\mathcal{T}$.

In order to compute $\ell := \mathsf{LCE}_T(i, i')$, we run $\texttt{lce}(v, v')$, where v and v' are the highest nodes such that $v.\mathsf{pos} = i$ and $v'.\mathsf{pos} = i'$, respectively. Such nodes can be obtained in $\mathcal{O}(r)$ time by initializing v and v' to be the root of $\mathcal{T}$ and repeatedly setting $v.\texttt{child}(v.\texttt{index}(i))$ or $v'.\texttt{child}(v'.\texttt{index}(i'))$ as long as $v.\mathsf{pos} < i$ or $v'.\mathsf{pos} < i'$, respectively. It remains to analyze the time of evaluating $\mathsf{LCE}_T(i, i')$. For this, we say that nodes (v, v') form a *matching pair* if $v.\mathsf{symb} = v'.\mathsf{symb}$ and $T[\,i\,..\,v.\mathsf{pos}\,) = T[\,i'\,..\,v'.\mathsf{pos}\,)$. Denote by N and N' the set of nodes v and v' participating in matching pairs; these pairs form a perfect matching between N and N'. We claim that the query algorithm satisfies the following additional invariant: No proper ancestor of v belongs to N and no proper ancestor of v' belongs to N'. The invariant is satisfied at the beginning because all proper ancestors of v and v' have their expansions starting before position i and i', respectively. In cases 2 and 4, the node v does not form a matching pair with any proper ancestor of v' (they do not belong to N' by our invariant), with v' itself (because $v.\mathsf{symb} \neq v'.\mathsf{symb}$), nor with any proper descendant of v' (their symbols are shorter than $\|v'.\mathsf{symb}\| \leq \|v.\mathsf{symb}\|$). Thus, v does not belong to N. By symmetry, the invariant remains satisfied in cases 2 and 3 when we replace v' by its leftmost child. In cases 5 and 6, proper ancestors of the nodes μ and μ' are proper ancestors of the nodes v and v'.

Let $\hat{N}$ consist of nodes that do not belong to N yet have descendants in N, and define $\hat{N}'$ analogously based on N' instead of N. As we trace the value v in subsequent recursive calls (including the evaluation of the $\cdot.\texttt{next}()$ function), we obtain an Euler

tour traversal of a subtree of $\mathcal{T}$ consisting of selected nodes in $\hat{N}$, their children, and possible the ancestors of $\texttt{leaf}(i+\ell)$ (if $\texttt{leaf}(i+\ell) \neq \textbf{nil}$) Similarly, v' visits selected nodes in $\hat{N}'$, their children, and possibly the ancestors of $\texttt{leaf}(i'+\ell)$ (if $\texttt{leaf}(i'+\ell) \neq \textbf{nil}$). Moreover, Case 6 guarantees that, for any node in $\hat{N}$ or $\hat{N}'$, we visit at most two among its children in N or N', respectively. Thus, the total runtime of the algorithm is $\mathcal{O}(|\hat{N}| + |\hat{N}'| + r)$.

It remains to prove that $|\hat{N}| = \mathcal{O}(r)$ and, by symmetry, $|\hat{N}'| = \mathcal{O}(r)$. For this, observe that Lemma 19 applied to $T[\,i\,..\,i+\ell\,) = T[\,i'\,..\,i'+\ell\,)$ implies that the nodes participating in the popped sequence $\mathsf{PSeq}(T[\,i\,..\,i+\ell\,))$ belong to N, so $\hat{N}$ may only contain proper ancestors of these nodes. By Definition 17, the nodes in the popped sequence have $\mathcal{O}(r)$ parents in total (since nodes within each L_k and R_k have a single parent). These nodes also have $\mathcal{O}(r)$ proper ancestors because the parse tree is of height $\mathcal{O}(r)$ and does not contain degree-one vertices. □

We remark that a representation of T as an r-round restricted recompression RLSLP constitutes an analogous representation of the reverse text $\overline{T}$ (it suffices to reverse the right-hand side of every production), so Proposition 21 can also be used to answer the following queries $\overline{\mathsf{LCE}}_T$ queries:

$$\overline{\mathsf{LCE}}_T(i,i') := \max\{d \in [\,0\,..\,\min(i,i')\,] : T[\,i-d\,..\,i\,) = T[\,i'-d\,..\,i'\,)\}.$$

4 Our IPM Query Algorithm

This section constitutes the proof of Theorem 1. In other words, we show how to answer IPM queries in $O(r)$ time provided that the text $T \in \Sigma^n$ is represented using an aribtrary r-round recompression RLSLP $\mathcal{G}$ (see Definition 8).

Recall that the query algorithm is given two fragments X and Y of the text T satisfying $|Y| < 2|X|$, and the goal is to identify exact occurrences of X contained within Y. Our $\mathcal{O}(r)$-time query algorithm performs the following steps:

1. Define $\ell = \max\{k \in \mathbb{Z}_{\geq 0} : |\bar{X}_k| > k\}$, where $(\bar{X}_k)_{k=0}^{\infty}$ is introduced in Definition 17 along with the popped sequence $\mathsf{PSeq}(X)$. As proved in Section 4.1, $\mathsf{rle}(\bar{X}_\ell)$ can be constructed in $\mathcal{O}(r)$ time.
2. Define an appropriate fragment Y'_ℓ of T_ℓ such that every occurrence of X within Y yields an occurrence of $\bar{X}_\ell$ within Y'_ℓ. As proved in Section 4.2, we can ensure that $|Y'_\ell| = \mathcal{O}(|\bar{X}_\ell|)$ and $\mathsf{rle}(Y'_\ell)$ can be constructed in $\mathcal{O}(r)$ time.
3. Use pattern matching in run-length encoded strings to find the occurrences of $\bar{X}_\ell$ in Y'_ℓ, represented as $\mathcal{O}(1)$ arithmetic progressions; see Section 4.3.
4. For each progression, use $\mathcal{O}(1)$ LCE queries (in $\mathcal{O}(r)$ time each) to find the occurrences of $\bar{X}_\ell$ in Y'_ℓ that extend to occurrences of X in Y; see Section 4.4.

4.1 Constructing the Proxy Pattern $\bar{X}_\ell$

Observe that, for every non-empty fragment X of T, we have $|\bar{X}_0| = |X| > 0$ and $|\bar{X}_k| = 0$ for $k \in \mathbb{Z}_{>r}$, so $\ell := \max\{k \in \mathbb{Z}_{\geq 0} : |\bar{X}_k| > k\}$ is well-defined.

Lemma 22 *The level* $\ell := \max\{k \in \mathbb{Z}_{\geq 0} : |\bar{X}_k| > k\}$ *and the run-length encoding* $\mathsf{rle}(\bar{X}_\ell)$ *of the proxy pattern can be constructed in* $\mathcal{O}(r)$ *time.*

Proof Let us first focus on computing the level ℓ. For this, we use Lemma 20 to compute $\mathsf{rle}(\mathsf{PSeq}(X))$ decomposed as $\mathsf{rle}(L_0)\cdots\mathsf{rle}(L_q)\cdot\mathsf{rle}(R_q)\cdots\mathsf{rle}(R_0)$, where $q = \max\{k \in \mathbb{Z}_{\geq 0} : \bar{X}_k \neq \varepsilon\}$.

For $k \in \mathbb{Z}_{\geq 0}$, define M_k to be the *multiset* of symbols appearing in $\bar{X}_k$, that is, $\{\bar{X}_k[\,i\,] : i \in [\,0\,..\,|\bar{X}_k|\,)\}$. Our strategy to iterate over levels $k \in [\,0\,..\,q\,]$ in the decreasing order maintaining a multiset M such that $M = M_k$ holds as soon as we complete processing level k. Initially, we set $M := \emptyset = M_{q+1}$. Thus, for each level k, our goal is to transform M_{k+1} into M_k. For this, we need to insert symbols present in L_k and R_k. Additionally, we need to expand symbols of $\bar{X}_{k+1}$ created by shrink_{k+1}. Thus, for each symbol $A \in M$ with $\mathsf{lvl}(A) = k+1$, we replace A with symbols of $\mathsf{rhs}(A)$.

In order to efficiently implement these operations, we store M as a monotone bucket priority queue [25], where the priority of each symbol A is its level $\mathsf{lvl}(A)$. In other words, we maintain an array $Q[\,0\,..\,q\,]$, where $Q[\,k'\,]$ is a list of symbols $A \in M$ such that $\mathsf{lvl}(A) = k'$ (each with the same multiplicity as in M). Upon inserting an element to A to M, we simply append it to the list $Q[\,\mathsf{lvl}(A)\,]$. Moreover, in order to retrieve all $A \in M$ with $\mathsf{lvl}(A) = k+1$, we iterate over the list $Q[\,k+1\,]$. For each such symbol A, our algorithm removes A from M and inserts all symbols of $\mathsf{rhs}(A)$ into M.

Recall that our goal is to identify the largest k such that $|\bar{X}_k| > k$ or, equivalently, $|M_k| > k$. Furthermore, observe that every operation on the set M increases the size $|M|$ — whenever we remove $A \in M$, we immediately insert $|\mathsf{rhs}(A)| \geq 2$ symbols of $\mathsf{rhs}(A)$. Consequently, it suffices to process subsequent integers k from q down to 0 and, while transforming M from M_{k+1} to M_k, stop as soon as $|M| > k$; we are then guaranteed that $|M_k| \geq |M| > k$. Since the stopping condition has not been satisfied at higher levels, we can also conclude that $\ell = k$ holds in this case.

As far as the running time of the algorithm is concerned, note that constructing $\mathsf{rle}(\mathsf{PSeq}(X))$ takes $\mathcal{O}(r)$ time. Initializing the array Q takes $\mathcal{O}(q) = \mathcal{O}(r)$ time. Finally, note that each operation on M takes constant time and the number of operations on M does not exceed $3|M|$ (because every deletion is immediately followed by at least two insertions). Consequently, since $|M| \leq \ell + \mathcal{O}(1) = \mathcal{O}(r)$ holds when the algorithm terminates, we conclude that the main phase of the algorithm also takes $\mathcal{O}(r)$ time.

In the next step, we construct $\bar{X}_{\ell+1}$. For this, it suffices to start with $L_{\ell+1}\cdots L_q \cdot R_q \cdots R_{\ell+1}$ and exhaustively expand every symbol at a level higher than $\ell+1$. Our implementation processes subsequent symbols in $L_{\ell+1}\cdots L_q \cdot R_q \cdots R_{\ell+1}$ from left to right using a recursive procedure that, for a given symbol A, either outputs A (if $\mathsf{lvl}(A) \leq \ell+1$) or recursively processes symbols of $\mathsf{rhs}(A)$ in the left-to-right order (otherwise). Since each production consists of at least two symbols, the total running time of this step is $\mathcal{O}(q - \ell + |\bar{X}_{\ell+1}|) = \mathcal{O}(r)$.

Finally, in order to determine $\mathsf{rle}(\bar{X}_\ell)$, we expand every symbol A in $\bar{X}_{\ell+1}$ with $\mathsf{lvl}(A) = \ell+1$ into $\mathsf{rle}(\mathsf{rhs}(A))$. We then prepend $\mathsf{rle}(L_\ell)$, append $\mathsf{rle}(R_\ell)$, and post-process the resulting sequence so that two powers of the same symbol are merged

whenever they appear next to each other. This step takes $\mathcal{O}(1+|\bar{X}_{\ell+1}|)=\mathcal{O}(r)$ time because $|\mathsf{rle}(\mathsf{rhs}(A))|\le 2$ holds for every symbol A. □

4.2 Constructing the Proxy Text Y'_ℓ

The following Definition 23 defines the appropriate proxy text Y'_ℓ as a fragment of T_ℓ such that $\mathsf{exp}(Y'_\ell)$ covers a sufficiently large fragment of Y. A natural choice would be to pick Y'_ℓ so that $\mathsf{exp}(Y'_\ell)$ covers the entire Y; unfortunately, this may result in a proxy text Y'_ℓ that is much longer than the proxy pattern X'_ℓ, even with respect to run-length encoding: the fact that X is covered by a length-$\mathcal{O}(\ell)$ fragment of $T_{\ell+1}$ does not imply that the same is true for Y. Nevertheless, if X occurs in Y, the fragment of Y containing all the occurrences of X is guaranteed to be covered by a length-$\mathcal{O}(\ell)$ fragment of $T_{\ell+1}$. Our definition assures that Y'_ℓ is long enough so that every occurrence of X in Y yields an occurrence of $\bar{X}_\ell$ in Y'_ℓ and, at the same time, short enough to enable efficient implementation of our query algorithm. Specifically, we define Y'_ℓ so that it covers the middle position Y (which is contained in every occurrence of X in Y) and extends in both directions just enough to cover, for every occurrence X containing that middle position, the induced occurrence of $\bar{X}_\ell$ in T_ℓ. We quantify this extension using the length (so that the occurrences of $\bar{X}_\ell$ in Y'_ℓ form $\mathcal{O}(1)$ arithmetic progressions; see Section 4.3) and the number of runs (so that $\mathsf{rle}(Y'_\ell)$ can be constructed efficiently; see Lemma 26). We also ensure that $\mathsf{exp}(Y'_\ell)$ does not extend beyond Y.

Definition 23 Let $T[\,m\,]$ be the middle character in Y and let $T_\ell[\,m_\ell\,]$ be its ancestor in T_ℓ. Moreover, let $e\in[\,m_\ell\,..\,|T_\ell|\,)$ be the largest position such that $|T_\ell[\,m_\ell\,..\,e\,]|\le|\bar{X}_\ell|+\ell$ and $|\mathsf{shrink}_{\ell+1}(T_\ell[\,m_\ell\,..\,e\,])|\le 2\ell+3$; symmetrically, let $b\in[\,0\,..\,m_\ell\,]$ be the smallest position such that $|T_\ell[\,b\,..\,m_\ell\,]|\le|\bar{X}_\ell|+\ell$ and $|\mathsf{shrink}_{\ell+1}(T_\ell[\,b\,..\,m_\ell\,])|\le 2\ell+3$. Define the *proxy text* Y'_ℓ as the maximal fragment of $T_\ell[\,b\,..\,e\,]$ whose expansion $\mathsf{exp}(Y'_\ell)$ is a fragment of T contained within Y.[7]

Lemma 24 *For every occurrence of X contained in Y, the corresponding occurrence of the proxy pattern $\bar{X}_\ell$ in T_ℓ stipulated by Lemma 19 is contained within the proxy text Y'_ℓ.*

Proof Let us fix an occurrence $T[\,i\,..\,j\,]$ of X contained in Y. For every $k\in[\,0\,..\,\ell\,]$, Lemma 19 yields a corresponding occurrence $T_k[\,i_k\,..\,j_k\,]=\bar{X}_k$. Let us also define $T_k[\,i'_k\,]$ as the ancestor of $T[\,i\,]$ in T_k. □

Claim 25 For every $k\in[\,0\,..\,\ell\,]$, we have $|T_k[\,i'_k\,..\,i_k\,)|\le k$.

Proof We proceed by induction. The claim is trivial for $k=0$ due to $i'_0=i_0=i$. For $k>0$, Lemma 19 implies $T_k[\,0\,..\,i_k\,)=\mathsf{shrink}_k(T_{k-1}[\,0\,..\,i_{k-1}\,)\cdot L_{k-1})$. In particular, $T_k[\,i'_k\,..\,i_k\,)=\mathsf{shrink}_k(L'_{k-1}\cdot T_{k-1}[\,i'_{k-1}\,..\,i_{k-1}\,)\cdot L_{k-1})$, where L'_{k-1} consists of the children of $T_k[\,i'_k\,]$ to the left of $T_{k-1}[\,i'_{k-1}\,]$. Observe that shrink_k does not place block boundaries within $L'_{k-1}\cdot T_{k-1}[\,i'_{k-1}\,]$ (all those symbols have $T_k[\,i'_k\,]$ as their parent) and within L_{k-1} (by Definition 17, L_{k-1} is either empty or the leftmost

[7] The proxy text is empty if there is no character in $T_\ell[\,b\,..\,e\,]$ whose expansion is contained within Y.

block of $\bar{X}_{k-1}$). The remaining possible locations for block boundaries within $L'_{k-1} \cdot T_{k-1}[\, i'_{k-1} \,..\, i_{k-1}\,) \cdot L_{k-1}$ are immediately after the characters of $T_{k-1}[\, i'_{k-1} \,..\, i_{k-1}\,)$, and the number of such locations does not exceed $|T_{k-1}[\, i'_{k-1} \,..\, i_{k-1}\,)| \le k-1$ by the inductive hypothesis. Hence, the number of blocks that $L'_{k-1} \cdot T_{k-1}[\, i'_{k-1} \,..\, i_{k-1}\,) \cdot L_{k-1}$ is partitioned into (with respect to shrink_k) equals $|T_k[\, i'_k \,..\, i_k\,)| \le k$. □

The condition $|Y| < 2|X|$ implies that $T[\, i \,..\, j\,]$ contains the middle position of Y. Thus, $i \le m$ and $i'_\ell \le m_\ell$. Consequently, Claim 25 implies $i_\ell \le i'_\ell + \ell \le m_\ell + \ell$. Hence, we have $j_\ell = i_\ell + |\bar{X}_\ell| - 1 \le m_\ell + |\bar{X}_\ell| + \ell - 1$ and $|T_\ell[\, m_\ell \,..\, j_\ell\,]| \le |\bar{X}_\ell| + \ell$. Therefore, $T_\ell[\, m_\ell \,..\, j_\ell\,]$ is either a suffix of $\bar{X}_\ell$ or consists of the entire $\bar{X}_\ell$ preceded by at most ℓ symbols. Since $|\mathsf{shrink}_{\ell+1}(\bar{X}_\ell)| = |\mathsf{shrink}_{\ell+1}(L_\ell) \cdot \bar{X}_{\ell+1} \cdot \mathsf{shrink}_{\ell+1}(R_\ell)| \le |\bar{X}_{\ell+1}| + 2 \le \ell + 3$, we conclude that $|\mathsf{shrink}_{\ell+1}(T_\ell[\, m_\ell \,..\, j_\ell\,])| \le \ell + 3 + \ell = 2\ell + 3$.

Recall that $e \in [\, m_\ell \,..\, |T_\ell|\,]$ has been chosen as the largest index such that $|T_\ell[\, m_\ell \,..\, e\,]| \le |\bar{X}_\ell| + \ell$ and $|\mathsf{shrink}_{\ell+1}(T[\, m_\ell \,..\, e\,])| \le 2\ell + 3$. As proved above, the index j_ℓ satisfies both conditions, so we derive $j_\ell \le e$. A symmetric argument shows that $b \le i_\ell$, and thus $T_\ell[\, i_\ell \,..\, j_\ell\,]$ is contained within $T_\ell[\, b \,..\, e\,]$.

By Definition 17, the expansion $\mathsf{exp}(T_\ell[\, i_\ell \,..\, j_\ell\,])$ is contained within $T[\, i \,..\, j\,]$ and, by our assumption, $T[\, i \,..\, j\,]$ is contained within Y, so the expansion $\mathsf{exp}(T_\ell[\, i_\ell \,..\, j_\ell\,])$ is contained within Y. By construction, Y'_ℓ is the maximal fragment contained within $T_\ell[\, b \,..\, e\,]$ whose expansion $\mathsf{exp}(Y'_\ell)$ is contained within Y. Hence, $T_\ell[\, i_\ell \,..\, j_\ell\,]$ must be contained within Y'_ℓ.

We have now established the existence of a proxy text. Next, we show to construct it in $\mathcal{O}(r)$ time.

Lemma 26 *The run-length encoding* $\mathsf{rle}(Y'_\ell)$ *of the proxy text* Y'_ℓ *can be constructed in* $\mathcal{O}(r)$ *time.*

Proof First, we compute a pointer to $T[\, m\,]$ using Observation 10, and then we move up until we arrive at a node of $\overline{\mathcal{T}}$ representing a character $T_{\ell+1}[\, m_{\ell+1}\,]$ at level $\ell + 1$. Next, we generate the following fragment of $T_{\ell+1}$:

$$\hat{Y}_{\ell+1} := T_{\ell+1}[\, \max(0, m_{\ell+1} - 2\ell - 2) \,..\, \min(|T_{\ell+1}|, m_{\ell+1} + 2\ell + 1)\,).$$

To do this, move from $T_{\ell+1}[\, m_{\ell+1}\,]$ by $2\ell + 2$ positions in both directions. This phase is implemented in $\mathcal{O}(r + \ell + 1) = \mathcal{O}(r)$ time using Lemma 15 (to move to the right) and its symmetric counterpart (to move to the left).

To derive $\mathsf{rle}(Y'_\ell)$ from $\hat{Y}_{\ell+1}$, we replace A with $\mathsf{rle}(\mathsf{rhs}(A))$ for every symbol A with $\mathsf{lvl}(A) = \ell + 1$, and then we trim the resulting string to make sure that it contains at most $|\bar{X}_\ell| + \ell - 1$ symbols on either side of $T_\ell[\, m_\ell\,]$ and that its expansion does not exceed beyond Y. This phase costs $\mathcal{O}(\ell + 1)$ time, for a total of $\mathcal{O}(r)$ time for the entire algorithm. □

4.3 Finding the Occurrences of $\bar{X}_\ell$ in Y'_ℓ

As observed in [26–28], exact pattern matching in run-length encoded strings can be implemented in linear time. Below, we adapt this result to support representing the occurrences as arithmetic progressions.

Proposition 27 (see [28, Theorem 1]) *Given the run-length encodings* $\mathsf{rle}(P)$, $\mathsf{rle}(S)$ *of strings* $P, S \in \Sigma^+$, *the set* $\mathsf{Occ}(P, S) = \{i \in [0..|S| - |P|] : P = S[i..i + |P|)\}$, *represented by at most* $\min(|\mathsf{rle}(S)|, \lfloor |S|/|P| \rfloor)$ *arithmetic progressions with difference at most* $|P|$, *can be constructed in* $\mathcal{O}(|\mathsf{rle}(P)| + |\mathsf{rle}(S)|)$ *time.*

Proof None of the papers [26–28] considers the case of $|\mathsf{rle}(P)| = 1$, in which reporting $\mathsf{Occ}(P, S)$ explicitly might be too costly. In this case, every occurrence of P is contained within a single run of S. A run $S[j..j + q) = b^q$ contains an occurrence of $P = a^p$ if and only if $a = b$ and $p \leq q$; then, the occurrences form an arithmetic progression $(j, j + 1, \ldots, j + q - p)$ with difference one. Each such progression corresponds to a run of length at least $|P|$ in S, so their number does not exceed $\min(|\mathsf{rle}(S)|, \lfloor |S|/|P| \rfloor)$. To construct them in $\mathcal{O}(p + s)$ time, we scan $\mathsf{rle}(S)$ maintaining the length of the already processed prefix of S.

For $|\mathsf{rle}(P)| \geq 2$, the algorithm of [28, Theorem 1] reports elements of $\mathsf{Occ}(P, S)$ one be one; the number of occurrences does not exceed $|\mathsf{rle}(S)| - 1$. By [1, Fact 1.1], if we greedily group occurrences into arithmetic progressions with a difference at most $|P|$ each, we obtain at most $\lfloor |S|/|P| \rfloor$ progressions in total. □

Applying Proposition 27 to $\bar{X}_\ell$ and Y'_ℓ, we obtain the following result:

Corollary 28 *Given* $\mathsf{rle}(\bar{X}_\ell)$ *and* $\mathsf{rle}(Y'_\ell)$, *the occurrences of* $\bar{X}_\ell$ *in* Y'_ℓ, *represented as* $\mathcal{O}(1)$ *arithmetic progressions, each with a difference of at most* $|\bar{X}_\ell|$, *can be computed in* $\mathcal{O}(r)$ *time.*

Proof By Lemmas 22 and 26, both $\mathsf{rle}(\bar{X}_\ell)$ and $\mathsf{rle}(Y'_\ell)$ are of size $\mathcal{O}(r)$. Moreover, the definition of Y'_ℓ (see Definition 23) guarantees that $|Y'_\ell| < 2|\bar{X}_\ell| + 2\ell < 4|\bar{X}_\ell|$, so the algorithm of Proposition 27 returns at most four progressions. □

4.4 Verifying Candidate Occurrences

Lemma 24 shows that every occurrence of X in Y corresponds to an occurrence of $\bar{X}_\ell$ in Y'_ℓ. Moreover, by Corollary 28, $\mathsf{Occ}(\bar{X}_\ell, Y'_\ell)$ is the union of $\mathcal{O}(1)$ arithmetic progressions, each with a difference of at most $|\bar{X}_\ell|$.

Let $V_\ell := \{\overline{a}_\ell + i \cdot g_\ell : i \in [0..s_\ell)\}$ be one such arithmetic progression. Observe that $Y'_\ell[\overline{a}_\ell + (i-1) \cdot g_\ell .. \overline{a}_\ell + i \cdot g_\ell) = \bar{X}_\ell[0..g_\ell)$ holds for each $i \in [1..s_\ell)$. Thus, the arithmetic progression V_ℓ of positions in Y'_ℓ corresponds to an arithmetic progression $V = \{\overline{a} + i \cdot g : i \in [0..s)\}$ of positions in Y, where $s = s_\ell$, $g = |\mathsf{exp}(\bar{X}_\ell[0..g_\ell))|$, and $Y[\overline{a}]$ is the leftmost leaf in the subtree of $\overline{\mathcal{T}}$ rooted at $Y'_\ell[\overline{a}_\ell]$. Since the non-terminals store their expansion lengths, a single pass over $\mathsf{rle}(\bar{X}_\ell)$ and $\mathsf{rle}(Y'_\ell)$ lets us compute the values g and $\overline{a}$, respectively.

Consequently, we henceforth assume that each arithmetic progression $V_\ell \subseteq \mathsf{Occ}(\bar{X}_\ell, Y'_\ell)$ is already represented by the underlying arithmetic progression V of positions in Y; note that each such progression V consists of starting positions of occurrences of $\mathsf{exp}(\bar{X}_\ell)$ in Y, and its difference g satisfies $g \leq |\mathsf{exp}(\bar{X}_\ell)|$.

Define $\overline{c} = |\mathsf{exp}(L_0 \cdots L_{\ell-1})|$ and $c = |X| - |\mathsf{exp}(R_{\ell-1} \cdots R_0)|$ so that $\mathsf{exp}(\bar{X}_\ell) = X[\,\overline{c}\,..\,c\,)$. Let $V = \{\overline{a} + i \cdot g : i \in [\,0\,..\,s\,)\}$ be any of the aforementioned arithmetic progressions representing occurrences of $\mathsf{exp}(\bar{X}_\ell)$ in Y. Our goal is to test, in bulk, for each position $\overline{a} + i \cdot g \in V$, whether X occurs in Y at position $\overline{a} + i \cdot g - \overline{c}$. Then, the only remaining step is to filter out occurrences of X that are not contained within Y. If $|V| = 1$, one can use a single LCE query to verify $Y[\,\overline{a} - \overline{c}\,..\,\overline{a} - \overline{c} + |X|\,) = X$. Thus, we henceforth assume $|V| > 1$.

Let $a = \overline{a} + (s-1) \cdot g + |\mathsf{exp}(\bar{X}_\ell)|$ be the position of Y immediately following the rightmost occurrence of $\mathsf{exp}(\bar{X}_\ell)$ captured by V. We ask the following LCE queries; see Figs. 5 and 6:

$$\overline{u} := \overline{\mathsf{LCE}}_X(\overline{c}, \overline{c} + g), \qquad u := \mathsf{LCE}_X(c, c - g),$$
$$\overline{v} := \overline{\mathsf{LCE}}_Y(\overline{a}, \overline{a} + g), \qquad v := \mathsf{LCE}_Y(a, a - g).$$

Case 1: $\overline{u} = \overline{c}$ and $u = |X| - c$. In this case, the period g of $\mathsf{exp}(\bar{X}_\ell) = X[\,\overline{c}\,..\,c\,)$ extends to the entire X. Therefore, an occurrence of $\mathsf{exp}(\bar{X}_\ell)$ contained in $Y[\,\overline{a}\,..\,a\,)$ extends to an occurrence of X in Y if and only if the latter is contained in $Y[\,\overline{a} - \overline{v}\,..\,a + v\,)$, which is the maximal extension of $Y[\,\overline{a}\,..\,a\,)$ with period g. The starting positions of these occurrences of X form an arithmetic progression:

$$\left\{\overline{a} - \overline{c} + i \cdot g : i \in \left[\left\lceil \frac{\max(0, \overline{u} - \overline{v})}{g} \right\rceil .. s - \left\lceil \frac{\max(0, u - v)}{g} \right\rceil\right)\right\}.$$

Case 2: $\overline{u} < \overline{c}$. Then, the period g of $\mathsf{exp}(\bar{X}_\ell) = X[\,\overline{c}\,..\,c\,)$ breaks at position $X[\,\overline{c} - \overline{u} - 1\,]$. The corresponding position within any occurrence of X in Y must also break the period. The period of $Y[\,\overline{a}\,..\,a\,)$ breaks at position $Y[\,\overline{a} - \overline{v} - 1\,]$. Thus, whenever $X[\,\overline{c}\,..\,c\,)$ is aligned within $Y[\,\overline{a}\,..\,a\,)$, then $X[\,\overline{c} - \overline{u} - 1\,]$ must be aligned against $Y[\,\overline{a} - \overline{v} - 1\,]$, and the only candidate occurrence is $Y[\,\overline{a} - \overline{c} + \overline{u} - \overline{v}\,..\,\overline{a} - \overline{c} + \overline{u} - \overline{v} + |X|\,)$. If this is not a valid fragment of Y (due to out-of-bounds indices), then there is no induced occurrence; otherwise, the single candidate can be verified with one LCE query.

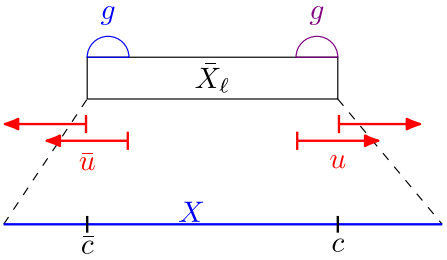


Fig. 5 The queries that compute $\overline{u}$ and u check how far the period g of $\mathsf{exp}(\bar{X}_\ell) = X[\,\overline{c}\,..\,c\,)$ extends within X

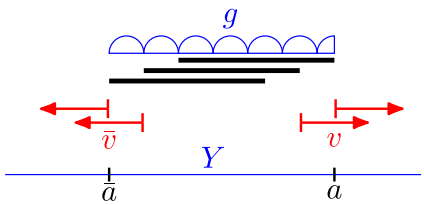


Fig. 6 The thick black lines are the detected occurrences of $\mathsf{exp}(\bar{X}_\ell)$ in Y; these occurrences start g positions apart and their union is $Y[\,\overline{a}\,..\,a\,)$. The queries that compute $\overline{v}$ and v check how far the period g of $Y[\,\overline{a}\,..\,a\,)$ extends within Y

Case 3: $u < |X| - c$.[8] This case is symmetric to the previous one up to reversing X and Y. The period g of $\mathsf{exp}(\bar{X}_\ell) = X[\,\overline{c}\,..\,c\,)$ breaks at position $X[\,c+u\,]$, whereas the period g of $Y[\,\overline{a}\,..\,a\,)$ breaks at position $Y[\,a+v\,]$. The only candidate occurrence is $Y[\,a-c+v-u\,..\,a-c+v-u+|X|\,)$, where these two positions are aligned; if this candidate is a valid fragment, it can be verified with a single LCE query.

Summary We were able to construct the set of occurrences of X in Y with up to five LCE queries for each of the $\mathcal{O}(1)$ arithmetic progressions representing $\mathsf{Occ}(\bar{X}_\ell, Y'_\ell)$. Each of these queries can be answered in $\mathcal{O}(r)$ time using Proposition 21. Therefore, our verification algorithm has a runtime of $\mathcal{O}(r)$.

Lemma 29 *Given* $\mathsf{Occ}(\bar{X}_\ell, Y'_\ell)$ *as a set of* $\mathcal{O}(1)$ *arithmetic progressions with differences at most* $|\bar{X}_\ell|$*, we can compute* $\mathsf{Occ}(X, Y)$ *in time* $\mathcal{O}(r)$.

We note that Lemma 29 is similar to [2, Lemma 1.14 (b)]. Both answer a set of LCE queries in a periodic string while only interested in the indices of the largest results. Since the LCE queries in our compressed setting are slower than in theirs, we get a larger runtime in total.

5 Proof-of-Concept Implementation

We implemented our algorithm in the Rust programming language and published our source code in the accompanying repository https://github.com/aduyster/ipm_in_log_time. This is a proof-of-concept implementation that follows our theoretical algorithm very closely and performs within the worst-case running time bounds. Our main objectives were to provide the first complete implementation of the PILLAR model in the compressed setting and to establish a baseline for future algorithm engineering research. In particular, practical optimizations and comprehensive experimental evaluation are beyond the scope of this work. Our implementation includes several benchmarks and, in this section, we report the results of our measurements.[9]

[8] If Cases 2 and 3 hold simultaneously, either procedure can be used.

[9] The benchmarks were carried out with a 13th Gen Intel(R) Core(TM) i5-1350P CPU (all measurements were run consecutively on a single core) and 32GB RAM (Memory speed: 5200 MT/s) and 32GB swap.

Our IPM query algorithm and the remaining PILLAR operations require the text to be represented as a (possibly restricted) recompression RLSLP. Thus, we also implemented a simple $O(n)$-time algorithm constructing recompression RLSLPs from the plain representation of the input text. Construction algorithms working directly on the compressed input text representation (such as an arbitrary SLP) exist in the literature [5, 6, 19, 20], and implementing them is an interesting future direction outside the scope of this study. With the current construction algorithms as a bottleneck, our evaluation is limited to texts of length up to $2^{32} \approx 4 \cdot 10^9$ characters.

In our preliminary experiments, we used finite prefixes of the infinite standard word with a constant directive sequence $(3, 3, \ldots)$ [29, Chapter 2]. Finite prefixes of this infinite string can be obtained through a recursive sequence $(S_i)_{i=1}^{\infty}$, where $S_0 := \mathtt{b}$, $S_1 := \mathtt{a}$, and $S_i := (S_{i-1})^3 \cdot S_{i-2}$ for $i \geq 2$. These strings are also known as 3-Fibonacci words since their recursive definition resembles the definition of the 3-Fibonacci sequence (describing the lengths of 3-Fibonacci words). The infinite 3-Fibonacci string is highly compressible and contains highly periodic substrings. Importantly for our experiments, a random IPM query has a constant probability of several outcomes: returning no occurrences, one occurrence, or at least two occurrences.

We asked IPM queries to texts T defined as prefixes $S[0..2^x)$ of the infinite 3-Fibonacci string S, with $x \in \{4, .., 32\}$. We picked queried fragments randomly (uniformly distributed over possible lengths and, for a fixed length, starting positions) and asked IPM queries for those.

In the plots, the horizontal axes represent the parameter x (equal to $\log_2 n$, i.e., logarithmic in text length) and the vertical axes denote runtime in microseconds. In all conditions, the graphs show linear increase, that is, the measured IPM queries take time logarithmic in the text length, matching our theoretical analysis.

In Fig. 7, we compare IPM queries that find any occurrences to those that do not. The former ones take longer on average, which can be easily explained by the fact that our query algorithm first generates some candidate occurrences and then verifies them. If there are no candidates, the query procedure terminates early.

In Fig. 8, we compared the performance of our query algorithm on two RLSLPs:

1. the restricted recompression RLSLPs built using the construction algorithm introduced in [2] and applied in [6, 21];
2. the classic (non-restricted) recompression RLSLPs introduced in [19, 20] and applied in the state-of-the-art dynamic strings implementation [7].

The restricted RLSLPs, even though smaller in the worst-case, typically have more levels due to the presence of inactive symbols, which makes the queries noticeably slower. It remains an interesting question whether a different choice of parameters (compared to the one used in [2, 6, 21] and our implementation) can reduce the number of levels without compromising the asymptotic guarantees on the total grammar size.

We tested on prefixes of the infinite 3-Fibonacci string. This string is highly compressible, in fact, our construction algorithm yielded data structures of size less than 6KB even for the string of length 2^{20}.

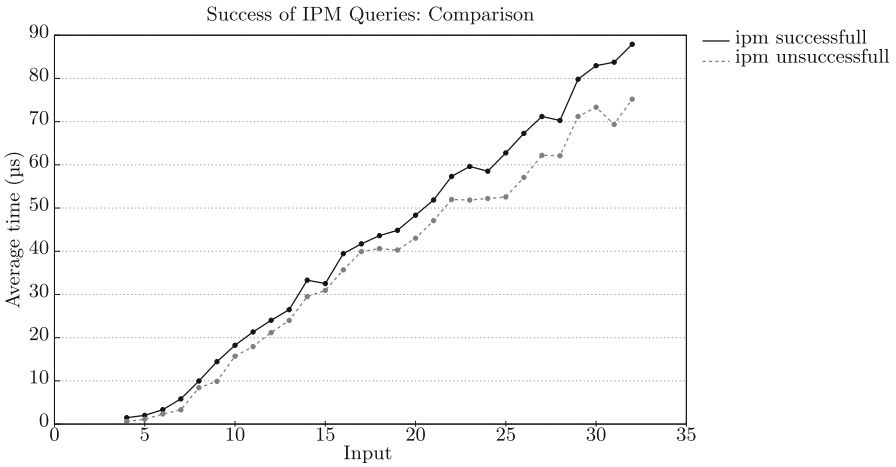


Fig. 7 The IPM query time grows logarithmically in string length. Queries that find no occurrence are faster than those that do. Measured on prefixes of the infinite 3-Fibonacci string

6 Conclusions

We presented an efficient algorithm for answering IPM queries on texts represented using (restricted) recompression RLSLPs, that is, run-length straight-line programs obtained using recompression or restricted recompression. Our algorithm takes $O(r)$ time, where r is the number of rounds that the underlying (restricted) recompression scheme needs to parse the text T; standard constructions achieve $r = \mathcal{O}(\log n)$.

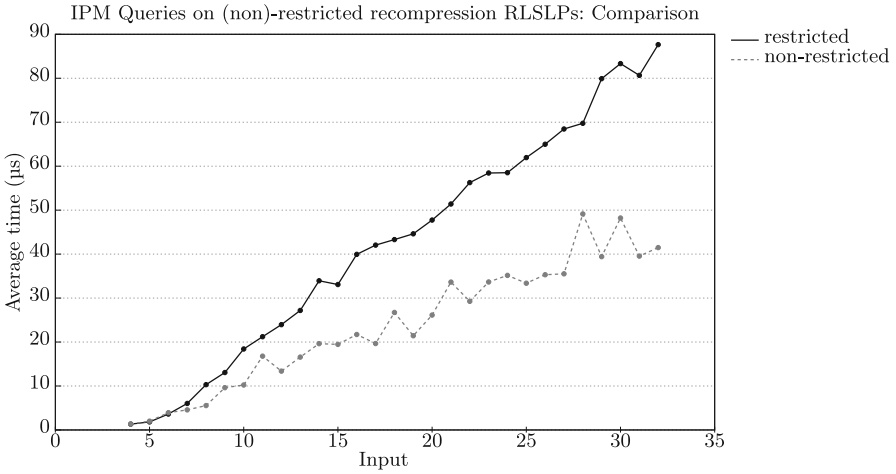


Fig. 8 The IPM query time grows logarithmically in string length. Queries on the restricted recompression RLSLP are slower than queries on the general recompression RLSLP. Measured on prefixes of the infinite 3-Fibonacci string

The provided RLSLP does not need to be preprocessed in any way; we only assume that every non-terminal stores its production, the length of its expansion, and in which round it has been created. This feature makes our IPM algorithm automatically applicable on top of any implementation of (restricted) recompression, without overhead in the construction time or space complexity. (The space allocated during the query algorithm is up to $O(r)$, and it can be freed once the query algorithm terminates.) In particular, our query algorithm is compatible with the compressed construction of [6] and the dynamic construction of [7]; see Corollaries 2 and 3. We also believe that it can also be used on top of the dynamic compressed implementation of restricted recompression from a recent preprint by Nishimoto and Tabei [30, Section 14].

This work eliminates the bottleneck in the PILLAR model in the compressed and dynamic settings. In the compressed setting, all PILLAR queries can be answered in $\mathcal{O}(\log n)$-time for a data structure of size $\mathcal{O}(\delta \log \frac{n}{\delta})$. It remains open whether this is query optimal and if a slightly larger data structure could improve query time further. In the dynamic setting it is unknown whether any other tradeoff exists between query time and update time.

Our Rust implementation demonstrates that our query algorithm does not hide any prohibitively complex components and works fast on synthetic compressible data. Nevertheless, further experimental evaluation on real-life strings is needed to guide heuristic optimizations that might lead to better query times in practice. To allow bigger datasets, implementing an efficient construction of the compressed data structure from the LZ representation is vital. With this and an implementation of the dynamic update procedure, PILLAR-model based algorithms can be compared to their non-PILLAR counterparts optimized for specific settings.

Acknowledgements Part of this work was done while Tomasz Kociumaka was affiliated with INSAIT, Sofia University "St. Kliment Ohridski", Bulgaria, funded from the Ministry of Education and Science of Bulgaria (support for INSAIT, part of the Bulgarian National Roadmap for Research Infrastructure).

Author Contributions Anouk Duyster and Tomasz Kociumaka jointly developed the main ideas and formalized them in the design and analysis of the resulting algorithm. A.D. implemented the prototype and performed the experiments, while T.K. assisted with debugging, refactoring, and experimental design. T.K. wrote the introductory and background sections, whereas A.D. prepared the technical sections; both authors proofread and approved the final manuscript. The work was carried out under T.K.'s supervision.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing Interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal Pattern Matching Queries in a Text and Applications. In: 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, P. Indyk (ed.), (SIAM, 2015), pp. 532–551. https://doi.org/10.1137/1.9781611973730.36
2. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. SIAM J. Comput. **53**(5), 1524–1577 (2024). https://doi.org/10.1137/23M1567618
3. Landau, G.M., Vishkin, U.: Fast string matching with k differences. J. Comput. Syst. Sci. **37**(1), 63–78 (1988). https://doi.org/10.1016/0022-0000(88)90045-1
4. Charalampopoulos, P., Kociumaka, T., Wellnitz, P.: Faster Approximate Pattern Matching: A Unified Approach, in 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Irani, S. (ed.), (IEEE Computer Society), pp. 978–989 (2020). https://doi.org/10.1109/FOCS46700.2020.00095
5. T. I.: Longest Common Extensions with Recompression. In: 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, LIPIcs, vol. 78, Kärkkäinen, J., Radoszewski, J., Rytter, W. (eds.), (Schloss Dagstuhl-Leibniz-Zentrum für Informatik), pp. 18:1–18:15 (2017). https://doi.org/10.4230/LIPIcs.CPM.2017.18
6. Kempa, D., Kociumaka, T.: Collapsing the Hierarchy of Compressed Data Structures: Suffix Arrays in Optimal Compressed Space. In: 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023 (IEEE), pp. 1877–1886 (2023). https://doi.org/10.1109/FOCS57990.2023.00114
7. Gawrychowski, P., Karczmarz, A., Kociumaka, T., Łącki, J., Sankowski, P.: Optimal Dynamic Strings, in 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, Czumaj, A. (ed.), (SIAM), pp. 1509–1528 (2018). https://doi.org/10.1137/1.97816119750 31.99
8. Cardinal, J., Iacono, J.: Modular Subset Sum, Dynamic Strings, and Zero-Sum Sets. in 4th Symposium on Simplicity in Algorithms, SOSA 2021, Le, H.V., King, V. (eds.), (SIAM), pp. 45–56 (2021). https://doi.org/10.1137/1.9781611976496.5
9. Charalampopoulos, P., Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T., Zuba, W.: Approximate Circular Pattern Matching, in 30th Annual European Symposium on Algorithms, ESA 2022, LIPIcs, vol. 244, Chechik, S., Navarro, G., Rotenberg, E., Herman, G. (eds.), (Schloss Dagstuhl-Leibniz-Zentrum für Informatik), pp. 35:1–35:19 (2022). https://doi.org/10.4230/LIPICS.ESA.2022.35
10. Charalampopoulos, P., Kociumaka, T., Wellnitz, P.: Faster Pattern Matching under Edit Distance: A Reduction to Dynamic Puzzle Matching and the Seaweed Monoid of Permutation Matrices. In: 63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022 (IEEE), pp. 698–707 (2022). https://doi.org/10.1109/FOCS54457.2022.00072
11. Charalampopoulos, P., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T., Zuba, W.: Approximate Circular Pattern Matching Under Edit Distance. In: 41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, LIPIcs, vol. 289, Beyersdorff, O., Kanté, M.M., Kupferman, O., Lokshtanov, D. (eds.), (Schloss Dagstuhl-Leibniz-Zentrum für Informatik), pp. 24:1–24:22 (2024). https://doi.org/10.4230/LIPICS.STACS.2024.24
12. Clifford, R., Gawrychowski, P., Kociumaka, T., Martin, D.P., Uznański, P.: The Dynamic k-Mismatch Problem, in 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, LIPIcs, vol. 223, Bannai, H., Holub, J (eds.), (Schloss Dagstuhl-Leibniz-Zentrum für Informatik), pp. 18:1–18:15 (2022). https://doi.org/10.4230/LIPICS.CPM.2022.18
13. Sahinalp, S.C., Vishkin, U.: Symmetry breaking for suffix tree construction. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23–25 May 1994, Montréal, Québec, Canada, Leighton, F.T., Goodrich, M.T. (eds.), (ACM), pp. 300–309 (1994). https://doi.org/10.1145/195058.195164
14. Sahinalp, S.C., Vishkin, U.: On a Parallel-Algorithms Method for String Matching Problems. In: Algorithms and Complexity, Second Italian Conference, CIAC '94, Rome, Italy, February 23–25, 1994, Proceedings, Lecture Notes in Computer Science, vol. 778, Bonuccelli, M.A, Crescenzi, P., Petreschi, R. (eds.), (Springer), pp. 22–32 (1994). https://doi.org/10.1007/3-540-57811-0_3
15. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997). https://doi.org/10.1007/BF02522825
16. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts, in 11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000, Shmoys, D.B. (ed.), (ACM/SIAM), pp. 819–828 (2000). http://dl.acm.org/citation.cfm?id=338219.338645

17. Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Fully Dynamic Data Structure for LCE Queries in Compressed Space, in 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22–26, 2016 - Kraków, Poland, LIPIcs, vol. 58, Faliszewski, P., Muscholl, A., Niedermeier, R. (eds.), (Schloss Dagstuhl - Leibniz-Zentrum für Informatik), pp. 72:1–72:15 (2016). https://doi.org/10.4230/LIPICS.MFCS.2016.72
18. Kempa, D., Kociumaka, T.: Dynamic suffix array with polylogarithmic queries and updates. In: 54th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2022, Leonardi, S., Gupta, A. (eds.), (ACM), pp. 1657–1670 (2022). https://doi.org/10.1145/3519935.3520061
19. Jeż, A.: Faster fully compressed pattern matching by recompression. ACM Trans. Algorithms **11**(3), 20:1–20:43 (2015). https://doi.org/10.1145/2631920
20. Jeż, A.: Recompression: A simple and powerful technique for word equations. J. ACM **63**(1), 4:1–4:51 (2016). https://doi.org/10.1145/2743014
21. Kociumaka, T., Navarro, G., Prezza, N.: Towards a definitive compressibility measure for repetitive sequences. IEEE Trans. Inf. Theory **69**(4), 2074–2092 (2023). https://doi.org/10.1109/TIT.2022.3224382
22. Kempa, D., Kociumaka, T.: Resolution of the Burrows-Wheeler Transform Conjecture. In: 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Irani, S. (ed.), (IEEE Computer Society), pp. 1002–1013 (2020). https://doi.org/10.1109/FOCS46700.2020.00097
23. Duyster, A., Kociumaka, T.: Logarithmic-Time Internal Pattern Matching Queries in Compressed and Dynamic Texts. In: 31st International Symposium on String Processing and Information Retrieval, SPIRE 2024, LNCS, vol. 14899, Lipták, Z., de Moura, E.S., Figueroa, K., Baeza-Yates, R. (eds.), (Springer), pp. 102–117 (2024). https://doi.org/10.1007/978-3-031-72200-4_8
24. Kieffer, J.C., Yang, E.: Grammar-based codes: A new class of universal lossless source codes. IEEE Trans. Inf. Theory **46**(3), 737–754 (2000). https://doi.org/10.1109/18.841160
25. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox (Springer) (2008). https://doi.org/10.1007/978-3-540-77978-0
26. Amir, A., Benson, G.: Efficient Two-Dimensional Compressed Matching. In: IEEE Data Compression Conference, DCC 1992, Storer, J.A., Cohn, M. (eds.), (IEEE Computer Society), pp. 279–288 (1992). https://doi.org/10.1109/DCC.1992.227453
27. Amir, A., Landau, G.M., Vishkin, U.: Efficient pattern matching with scaling. Journal of Algorithms-cognition Informatics and Logic **13**(1), 2–32 (1992). https://doi.org/10.1016/0196-6774(92)90003-U
28. Chung, K.L.: Fast string matching algorithms for run-length coded strings. Computing **54**(2), 119–125 (1995). https://doi.org/10.1007/BF02238127
29. Lothaire, M.: Algebraic Combinatorics on Words. Encyclopedia of Mathematics and its Applications (Cambridge University Press) (2002)
30. Nishimoto, T., Tabei, Y.: Dynamic suffix array in optimal compressed space. arXiv:2404.07510

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.