Grammar Index By Induced Suffix Sorting *

Tooru Akagi Dominik Köppl Yuto Nakashima Shunsuke Inenaga
Hideo Bannai Masayuki Takeda

June 16, 2026

Abstract

Pattern matching is the most central task for text indexes. Many recent indexes leverage compression techniques to make pattern matching feasible for massive but highly-compressible datasets. Among this class of indexes, we propose a new compressed text index built upon a grammar compression based on induced suffix sorting [Nunes et al., DCC'18]. We show that this grammar exhibits a locality sensitive parsing property, which allows us to specify, given a pattern P, certain substrings of P, called *cores*, that are similarly parsed in the text grammar whenever these occurrences are extensible to occurrences of P. Supported by the cores, given a pattern of length m, we can locate all its occ occurrences in a text T of length n within $\mathcal{O}(m \lg |\mathcal{S}| + \mathrm{occ}_C \lg |\mathcal{S}| \lg n + \mathrm{occ})$ time, where $\mathcal{S}$ is the set of all characters and non-terminals, occ is the number of occurrences, and occ_C is the number of occurrences of a chosen core C of P in the right-hand sides of all production rules of the grammar of T. Our grammar index requires $\mathcal{O}(g)$ words of space and can be built in $\mathcal{O}(n)$ time using $\mathcal{O}(g)$ working space on top of the input text, where g is the sum of the lengths of the right-hand sides of all production rules. We underline the strength of our grammar index with an exhaustive practical evaluation that gives evidence that our proposed solution excels at locating long patterns in highly-repetitive texts. Our implementation is available at `https://github.com/TooruAkagi/GCIS_Index`.

Keywords: grammar compression, locality sensitive parsing, induced suffix sorting, text indexing data structure

1 Introduction

Although highly-repetitive texts are perceived as a common problem instance nowadays due to use cases involving large sets of DNA sequences within a very narrow taxonomy group or huge numbers of slightly different versions of documents maintained in revision control systems, this problem is difficult to handle due to the large amount of data, but interesting in the sense that the data is highly compressible. Compressed text indexes have become the standard tool for tackling this problem when full-text search queries like locating all occurrences of a pattern are of importance. When working on indexes on highly-repetitive data, a desired property is to have a *self-index*, i.e., a data structure that supports queries on the underlying text without storing the text in its plain form. Such self-indexes include *grammar indexes*. A grammar index is an augmentation of the admissible grammar [29] produced by a grammar compressor. Grammar indexes exhibit strong compression ratios for semi-automatically generated or highly-repetitive texts. Unlike other indexes that perform pattern matching stepwise character-by-character, some grammar indexes have locality sensitive parsing properties, which allow them to match certain non-terminals of the admissible grammar built upon the pattern with the non-terminals of the text. Such a property helps us to perform fewer comparisons, and thus speeds up pattern matching for particularly long patterns, which could be large gene sequences in a genomic database or source code files in a database maintaining source code. Here, our focus is set on indexes that support $\mathsf{locate}(P)$ queries retrieving the starting positions of all occurrences of a given pattern P in a given text.

*This article is an extension of a contribution [1] to the String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021.

Table 1: Space (in words) and query time needed for answering locate for a pattern of length m regarding the indexes addressed in this article, cf. Sect. 1.2.

Index	Space	Locate Time
Claude and Navarro [7]	$\mathcal{O}(\hat{g})$	$\mathcal{O}(m^2 \lg \lg_{\hat{g}} n + (m + \mathrm{occ}) \lg \hat{g})$
Gagie et al. [22]	$\mathcal{O}(\hat{g} + z \lg \lg z)$	$\mathcal{O}(m^2 + (m + \mathrm{occ}) \lg \lg n)$
Christiansen et al. [5]	$\mathcal{O}(\gamma \lg(n/\gamma))$	$\mathcal{O}(m + \lg^\epsilon \gamma + \mathrm{occ} \lg^\epsilon(\gamma \lg(n/\gamma)))$
Christiansen et al. [5]	$\mathcal{O}(\gamma \lg(n/\gamma) \lg^\epsilon(\gamma \lg(n/\gamma)))$	$\mathcal{O}(m + \mathrm{occ})$
Lyndon SLP [43]	$\mathcal{O}(g_{\mathrm{Lyn}})$	$\mathcal{O}(m + \lg m \lg n + \mathrm{occ} \lg g_{\mathrm{Lyn}})$
ESP [41]	$\mathcal{O}(g_{\mathrm{ESP}})$	$\mathcal{O}(\lg \lg g_{\mathrm{ESP}} (m + \mathrm{occ}_C \lg m \lg n) \lg^* n)$
Signature [34, 35]	$\mathcal{O}(g_{\mathrm{sig}})$	$\mathcal{O}(m \sqrt{\frac{\lg g_{\mathrm{sig}}}{\lg \lg g_{\mathrm{sig}}}} + \lg g_{\mathrm{sig}} \lg m \lg^* n (\lg n + \lg m \lg^* n) + \mathrm{occ} \lg n)$
GCIS (this work)	$\mathcal{O}(g)$	$\mathcal{O}(m \lg \lvert\mathcal{S}\rvert + \mathrm{occ}_C \lg n \lg \lvert\mathcal{S}\rvert + \mathrm{occ})$
r-index [23]	$\mathcal{O}(r \lg(n/r))$	$\mathcal{O}(m + \mathrm{occ})$

n is the length of T, z is the number of LZ77 [44] phrases of T, γ is the size of the smallest string attractor [28] of T, g_{Lyn} is the size of the Lyndon SLP of T, $\hat{g}$ is the size of a given admissible grammar, $\epsilon > 0$ is a constant, m is the length of a pattern P, g_{ESP} is the size of the ESP grammar of T, r is number of BWT runs, and occ is the number of occurrences of P in T.

1.1 Our Contribution

Our main contribution is the discovery of a locality sensitive parsing property in the grammar produced by the grammar compression by induced suffix sorting (GCIS) [37], which helps us to answer locate with an index built upon GCIS with the following bounds:

Theorem 1.1. Given a text T of length n, we can compute an indexing data structure on T in $\mathcal{O}(n)$ time, which can locate all occ occurrences of a given pattern of length m in $\mathcal{O}(m \lg |\mathcal{S}| + \mathrm{occ}_C \lg n \lg |\mathcal{S}| + \mathrm{occ})$ time, where $\mathcal{S}$ is the set of characters and non-terminals of the GCIS grammar and occ_C is the number of occurrences in the right-hand sides of the production rules of the GCIS grammar of a selected core of the pattern, where a core is a string of symbols of the grammar of P defined in Sect. 4.1. Our index uses $\mathcal{O}(g)$ words of working space, where g is the sum of the lengths of the right-hand sides of all production rules.

Similar properties hold for other grammars such as the signature encoding [34], ESP [10], HSP [21], the Rsync parse [24], or the grammar of Christiansen et al. [5, Sect. 4.2]. A brief review of these and other self-indexes follows.

1.2 Related Work

In the field of self-indexes, grammar compression has won high popularity since a grammar can eliminate repetitions while exhibiting powerful query properties. A grammar index is a *self-index*, i.e., a data structure that supports queries on the underlying text without storing the text in its plain form.

With respect to indexing a grammar for answering locate, the first work we are aware of is due to Claude and Navarro [6] who studied indexes built upon so-called *straight-line programs (SLPs)*. An SLP is a context-free grammar representing a single string in the Chomsky normal form. We present their complexity bounds along with improved versions [7, 9, 22] in Table 1.

Other research focused on particular types of grammar, such as the ESP-index [33, 41, 42], an index [8] combining Re-Pair [31] with the Lempel–Ziv-77 parsing [44], a dynamic index [35] based on signature encoding [34], the Lyndon SLP [43], or the grammar index of Christiansen et al. [5]. For the experiments in Sect. 8, we will additionally have a look at other self-indexes capable of locate-queries. There, we analyze Burrows–Wheeler-transform (BWT) [4]-based approaches, namely the FM-index [19] and the r-index [23].

Finally, the grammar GCIS has other interesting properties besides being locality sensitive. Crescenzi et al. [11] used a factorization similar to GCIS to sparsify the text for string matching and indexing in space of the sparsification. Nunes et al. [38] showed how to compute the suffix array and the longest-common-prefix array from GCIS during a decompression step restoring the

original text. Subsequently, Díaz-Domínguez and Navarro [13] showed how to compute the BWT directly from the GCIS grammar, while Deng et al. [12] presented a BWT-based index built on the text that has been compressed by the GCIS grammar. This idea was later used by Hong et al. [26] to accelerate count queries on the FM-index.

2 Preliminaries

With lg we denote the logarithm to base two (i.e., $\lg = \log_2$). Given two integers i, j, we denote the interval $[i..j] = \{i, i+1, \ldots, j-1, j\}$, with $[i..j] = \{\}$ if $i > j$. Our computational model is the standard word RAM with machine word size $\Omega(\lg n)$, where n denotes the length of a given input string $T[1..n]$, which we call *the text*, whose characters are drawn from an integer alphabet Σ of size $n^{\mathcal{O}(1)}$. We call the elements of Σ *characters*. For a string $S \in \Sigma^*$, we denote with $S[i..]$ its i-th suffix, and with $|S|$ its length. The order $<$ on the alphabet Σ induces a lexicographic order on Σ^*, which we denote by $\prec$. A table of all frequently used variable names and their meaning is depicted in Table 2 for reference.

Table 2: Symbol table listing all used variables and symbols.

Symbol	**Meaning**	**First Appearance**
n	Length of the text T	Sect. 2
T	Input text of length n	Sect. 2
Σ	Integer alphabet	Sect. 2
Γ	Set of non-terminals	Sect. 3
$\mathcal{S}$	Set of characters and non-terminals, $\mathcal{S} := \Sigma \cup \Gamma$	Sect. 2
m	Length of pattern P	Thm. 1.1
P	Pattern to search for in text T	Thm. 1.1
lg	Logarithm to base 2 ($\lg = \log_2$)	Sect. 2
$\prec$	Lexicographic order	Sect. 2
occ	Number of occurrences of pattern P in text T	Thm. 1.1
occ_C	Number of occurrences of core C in right-hand sides	Thm. 1.1
g	Sum of lengths of right-hand sides of all grammar rules	Thm. 1.1
$\mathcal{G}_T$	GCIS grammar of text T, $\mathcal{G}_T := (\Sigma, \Gamma, \pi, X^{(\tau_T)})$	Sect. 3
$\mathcal{G}_P$	GCIS grammar of pattern P	Sect. 4.1
π	Production function, $\pi : \Gamma \to (\Sigma \cup \Gamma)^+$	Sect. 3
π^*	Expansion function (iterative application of π)	Sect. 3
$\mathcal{T}_T$	Derivation tree of grammar $\mathcal{G}_T$	Sect. 3
GST	Generalized suffix tree	Sect. 3
DAG	Directed acyclic graph representation of grammar	Sect. 3
τ_T	Height of derivation tree $\mathcal{T}_T$	Sect. 3
τ_P	Height of derivation tree of pattern grammar $\mathcal{G}_P$	Sect. 4.1
$T^{(h)}$	String at height h in recursive construction	Sect. 3
$X^{(\tau_T)}$	Start symbol of grammar $\mathcal{G}_T$	Sect. 3
$X_i^{(h)}$	Non-terminal at height h with index i	Sect. 3
$F_i^{(h)}$	Factor i at height h in LMS factorization	Sect. 3
$Y^{(h)}$	Non-terminal at height h in matching algorithm	Sect. 4.2
$\Gamma^{(h)}$	Set of non-terminals at height h	Sect. 3
L	L-type suffix	Sect. 3
S	S-type suffix	Sect. 3
S^*	LMS suffix (S*-type)	Sect. 3
C	Core of pattern P	Sect. 4.1
C_p	Prefix part of core partitioning	Sect. 4.1

C_s	Suffix part of core partitioning	Sect. 4.1
D	Array storing extended non-terminals for LCE queries	Sect. 5
W	List of non-terminals with pattern occurrences	Sect. 4.2
R	Concatenation of all right-hand sides	Sect. 3
Q	Array of starting positions in R	Sect. 7
L	Array of expansion lengths	Sect. 3
F	Sequence of first symbols (Elias-Fano encoded)	Sect. 7
Δ	Delta-encoded right-hand sides	Sect. 7
lce	Longest common extension function	Sect. 3
child	Child node function in GST	Sect. 3
locate	Pattern location function	Introduction
lookup	Non-terminal lookup function	Sect. 3
string_depth	String depth function for GST nodes	Sect. 3
lca	Lowest common ancestor function	Sect. 3

2.1 Induced Suffix Sorting

SAIS [36] is a linear-time algorithm for computing the suffix array [32]. We briefly review the parts of SAIS important for constructing the GCIS grammar. SAIS assigns each suffix of length at least two a type, which is either `L` or `S`:

- $T[i..]$ is an `L` suffix if $T[i..] \succ T[i+1..]$, or
- $T[i..]$ is an `S` suffix otherwise, i.e., $T[i..] \prec T[i+1..]$.

Finally, we stipulate that the suffix $T[n..]$ of length one is always type `S`. Since it is not possible that $T[i..] = T[i+1..]$, SAIS assigns each suffix a type. An `S` suffix $T[i..]$ is additionally an `S`* suffix (also called LMS suffix in [36]) if $T[i-1..]$ is an `L` suffix. The substring between two succeeding `S`* suffixes is called an *LMS substring*. In other words, a substring $T[i..j]$ with $i < j$ is an LMS substring if and only if $T[i..]$ and $T[j..]$ are `S`* suffixes and there is no $k \in [i+1..j-1]$ such that $T[k..]$ is an `S`* suffix. Regarding the defined types, we make no distinction between suffixes and their starting positions (e.g., the statements that (a) $T[i]$ is type `L` and (b) $T[i..]$ is an `L` suffix are equivalent). In fact, we can determine `L` and `S` positions solely based on their succeeding positions with the equivalent definition:

- if $T[i] > T[i+1]$, then $T[i]$ is `L`;
- if $T[i] < T[i+1]$, then $T[i]$ is `S`;
- finally, if $T[i] = T[i+1]$, then $T[i]$ has the same type as $T[i+1]$.

In what follows, we define a factorization that splits the text into factors at each `S`* position. To have the property that each factor starts with an `S`* position, we prepend a special character `#` smaller than all characters appearing in T to the text T. Then splitting `#`T at all `S`* positions gives the factorization `#`$T = F_1 \cdots F_z$ such that each factor starts with a `S`* position and all `S`* positions of T are factor starting positions. This factorization is called *LMS-factorization*. The LMS-factorization is uniquely defined by the fact that the `S`* positions are the factor beginning positions. By replacing each factor F_i by the lexicographic rank of its respective LMS substring[1], we obtain a string $T^{(1)}$ of these ranks. We recurse on $T^{(1)}$ until we obtain a string $T^{(\tau_T - 1)}$ whose rank-characters are all unique or whose LMS-factorization consists of at most two factors.

[1]For SAIS to work, it uses a slightly different order on the LMS substrings, called LMS-order. It differs from the lexicographic order when comparing two LMS substrings, where one of them is a prefix of the other. In such a case, the LMS-order would give the longer string a smaller rank.

2.2 Constructing the Grammar

We assign each computed factor $F_j^{(h)}$ a non-terminal $X_j^{(h)}$ such that $X_j^{(h)} \to F_j^{(h)}$, but omit the delimiter `#`. The order of the non-terminals $X_j^{(h)}$ is induced by the lexicographic order of their respective LMS-substrings. We now use the non-terminals instead of the lexicographic ranks in the recursive steps. If we set $X^{(\tau_T)} \to T^{(\tau_T-1)}$ as the start symbol, we obtain a context-free grammar $\mathcal{G}_T := (\Sigma, \Gamma, \pi, X^{(\tau_T)})$, where Γ is the set of non-terminals and a function $\pi : \Gamma \to (\Sigma \cup \Gamma)^+$ that applies (production) rules. For simplicity, we stipulate that $\pi(c) = c$ for $c \in \Sigma$. Let g denote the sum of the lengths of the right-hand sides of all grammar rules. We say that a non-terminal ($\in \Gamma$) or a character ($\in \Sigma$) is a *symbol*, and denote the set of characters and non-terminals with $\mathcal{S} := \Sigma \cup \Gamma$. We understand π also as a string morphism $\pi : \mathcal{S}^* \to \mathcal{S}^*$ by applying π on each symbol of the input string. This allows us to define the *expansion* $\pi^*(X)$ of a symbol X, which is the iterative application of π until obtaining a string of characters, i.e., $\pi^*(X) \in \Sigma^*$ and $\pi^*(X^{(\tau_T)}) = T$. Since $\pi(X)$ is deterministically defined, we call $\pi(X)$ the *right-hand side* (*RHS*) of X, and identify a non-terminal X with its rule $X \to \pi(X)$ when convenient.

Lemma 2.1 ([37])**.** The GCIS grammar $\mathcal{G}_T$ can be constructed in $\mathcal{O}(n)$ time. $\mathcal{G}_T$ is *reduced*, meaning that we can reach all non-terminals of Γ from $X^{(\tau_T)}$.

$\mathcal{G}_T$ can be visualized by its derivation tree $\mathcal{T}_T$, which has $X^{(\tau_T)}$ as its root. Each rule $X_k^{(h)} \to X_i^{(h-1)} \cdots X_j^{(h-1)}$ defines a node $X_k^{(h)}$ having $X_i^{(h-1)}, \ldots, X_j^{(h-1)}$ as its children. The height of $\mathcal{T}_T$ is $\tau_T = \mathcal{O}(\lg n)$ because the number of LMS substrings of $T^{(h)}$ is at most half of the length of $T^{(h)}$ for each recursion level h. The leaves of $\mathcal{T}_T$ are the terminals at height 0 that constitute the characters of the text T. Reading the nodes on height $h \in [0..\tau_T - 1]$ from left to right gives $T^{(h)}$ with $T^{(0)} = T$. We use $\mathcal{T}_T$ only as a conceptual construct since it would take $\mathcal{O}(n)$ words of space. Instead, we merge (identical) subtrees of the same non-terminal together to form a directed acyclic graph DAG, which is implicitly represented by π as follows:

By construction, each non-terminal appears exactly in one height of $\mathcal{T}_T$. We can therefore separate the non-terminals into the sets $\Gamma^{(1)}, \ldots, \Gamma^{(\tau_T)}$ such that a non-terminal of height h belongs to $\Gamma^{(h)}$. More precisely, π maps a non-terminal on height $h \geq 1$ to a string of symbols on height $h-1$ (in particular: a string of characters for $h = 1$). Hence, the grammar is acyclic.

Unfortunately, there are strings with $g = \Theta(n)$, so for a particular input, the grammar does not compress at all: Let $T = \prod_{i=0}^{m} \mathtt{a}^i\mathtt{b} = \mathtt{b} \cdot \mathtt{ab} \cdot \mathtt{aab} \cdot \mathtt{aaab} \cdots$ with $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\mathtt{a} < \mathtt{b}$. Then we have the rules $X_i^{(1)} \to \mathtt{a}^i\mathtt{b}$ for $i \in [0..m]$, with $X^{(2)} \to \prod_{i=0}^{m} X_i^{(1)}$ being the (production) rule of the start symbol $X^{(2)}$. Hence, $\tau_T = 2$ and $g = |T| + |\pi(X^{(2)})| = |T| + m + 1$. Nevertheless, the experiments in Sect. 8 show that the grammar is suitable for most highly-repetitive text collections.

3 GCIS Index

In what follows, we want to show that we can augment $\mathcal{G}_T$ with auxiliary data structures for answering locate. Our idea stems from the classic pattern matching algorithm with the suffix tree, cf. the textbook solution of Gusfield [25, APL1]. The key difference is that we search the core of a pattern — we defer the definition of a core to Sect. 4.1 — in the RHSs of the rules of $\mathcal{G}_T$ instead of searching the pattern itself in the text T. For that, we make use of the generalized suffix tree GST built upon the RHSs of all rules separated by a special delimiter symbol `$` being smaller than all symbols. Specifically, we rank the rules such that $\{X_1, \ldots, X_{|\Gamma|}\} = \Gamma$ (this ranking will be fixed later), and set $R := \pi(X_1)\$\pi(X_2)\$\cdots\pi(X_{|\Gamma|})\$$. Since we have a budget of $\mathcal{O}(g)$ words, we can afford to use a plain pointer-based tree topology. The leaf λ with the string label $Y \cdots \$\pi(X_{j+1})\$\pi(X_{j+2})\$\cdots\pi(X_{|\Gamma|})\$$ for Y being a non-empty (but not necessarily proper) suffix of $\pi(X_j)$ stores a pointer to the non-terminal X_j and an offset o such that $\pi(X_j)[o..]$ is a prefix of λ's string label. Next, we need the following operations on GST: First, $\mathsf{lca}(u, v)$ gives the lowest common ancestor (LCA) of two nodes u and v. We can augment GST with the data structure of Buchsbaum et al. [3] in linear time and space in the number of nodes of GST. This data structure

answers lca in constant time. Next, child(u, c) gives the child of the node u connected to u with an edge having a label starting with $c \in \Gamma \cup \Sigma \cup \{\$\}$. Our GST implementation answers child in $\mathcal{O}(\lg |\mathcal{S}|)$ time. For that, each node stores the pointers to its children in a binary search tree with the first symbol of each connecting edge as key. Finally, string_depth(v) returns the string depth of a node v, i.e., the length of its string label, which is the string read from the edge labels on the path from the root to v. We can compute and store the string depth of each node during its construction. The operation child allows us to compute the *locus* of a string S, i.e., the highest GST node u whose string label has S as a prefix, in $\mathcal{O}(|S| \lg |\mathcal{S}|)$ time. For each $\pi(X)$, we augment the locus u of $\pi(X)\$$ with a pointer to X such that we can perform lookup(S) for a string of symbols $S \in \mathcal{S}^*$ returning the non-terminal X with $\pi(X) = S$ or an invalid symbol $\perp$ if such an X does not exist. The time is dominated by the time for computing the locus of S. Finally, all leaves in suffix order are stored in a linked list such that we can traverse the leaves in lexicographic order with respect to their corresponding suffixes.

Linkage to the Grammar Each non-terminal $X \in \Gamma$ stores an array $X.P$ of $|\pi(X)|$ pointers to the leaves in GST such that the $X.P[i]$ points to the leaf that points back to X and has offset i (its string label has $\pi(X)[i..]$ as a prefix). Additionally, each non-terminal X stores the length of $\pi(X)$, an array $X.L$ of all expansion lengths of all its prefixes, i.e., $X.L[i] := \sum_{j=1}^{i} |\pi^*(\pi(X)[j])|$, and an array $X.R$ of the lengths of the RHSs of all its prefixes, i.e., $X.R[i] := \sum_{j=1}^{i} |\pi(\pi(X)[j])|$.

LCE queries Each internal node v stores a pointer to the leftmost leaf in the subtree rooted at v. With that we can use the function lce(X, Y, i, j) returning the *longest common extension* (LCE) of $\pi(X)[i..]$ and $\pi(Y)[j..]$ for $X, Y \in \Gamma$ and $i \in [1..|\pi(X)|], j \in [1..|\pi(Y)|]$. We can answer lce$(X, Y, i, j)$ by selecting the leaves $X.P[i]$ and $Y.P[j]$, retrieve the LCA lca$(X.P[i], Y.P[j])$ of both leaves, and take its string depth, all in constant time. More strictly speaking, we return $\min(|\pi(X)[i..]|, |\pi(Y)[j..]|,$ string_depth(lca$(X.P[i], Y.P[j])))$, since the delimiter \$ is not a unique character, but appears at each end of each RHS in the underlying string R of GST.

Complexity Bounds GST can be computed in $\mathcal{O}(g)$ time [18]. The grammar index consists of the GCIS grammar, GST built upon $|R| = g + |\Gamma|$ symbols, and augmented with a data structure for lca [3]. This all takes $\mathcal{O}(g)$ space. Each non-terminal is augmented with an array $X.P$ of pointers to leaves, $X.L$ and $X.R$ storing the expansion lengths of all prefixes of $\pi(X)$, which take again $\mathcal{O}(g)$ space when summing over all non-terminals.

4 Pattern Matching Algorithm

We follow the steps of Sahinalp and Vishkin [40, Sect. 2], who identify a substring C of the parse of the pattern P that can be matched against the RHSs of the grammar. They called such a substring C a *core* of the pattern. The core needs to exhibit certain properties such that all occurrences of P in T are parsed by the grammar of T in a way that their contained parse of C is identical. This allows us to retrieve the occurrences of P by (1) first finding the occurrences of C in the RHSs of the grammar and then (2) extending these occurrences to occurrences of P. However, we do not try to find the occurrences of P directly, but rather the lowest DAG nodes (a) having C as a descendant and (b) having such a large expansion that we could extend C to P.

4.1 Cores

Central to our pattern matching algorithm is the concept of *cores*. A core $C \in \left(\Gamma^{(h)}\right)^+$ with $\Gamma^{(0)} := \Sigma$ is a string of symbols at a certain height $h \geq 0$ of the GCIS grammar $\mathcal{G}_P$ built on the pattern P with the following properties.

- The expansion $\pi^*(C)$ of C is a substring of P.

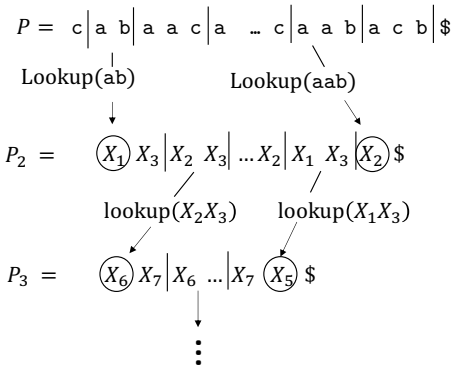

Figure 1: Computing $\mathcal{G}_P$ in Sect. 4.1. For each height h, we compute the LMS factorization $P^{(h)} = F_1^{(h)} \cdots F_z^{(h)}$ of $P^{(h)}$, query $Y_i^{(h+1)} = \mathsf{lookup}(F_i^{(h)})$ for $i \in [2..z-1]$, and continue with $P^{(h+1)} := \prod_{i=2}^{z_h-1} Y_i^{(h+1)}$ unless there is one i with $Y_i^{(h+1)} = \bot$, for which we abort the pattern matching.

- Given an occurrence of $\pi^*(C)$ is covered by consecutive nodes on a height $h \geq 0$ in $\mathcal{T}_T$ such that every node's expansion covers at least one character of $\pi^*(C)$, then this occurrence of $\pi^*(C)$ is *not* part of an occurrence of P in T if these nodes do not have all the same parent node on height $h+1$. Technically speaking, we consider symbols $X_i^{(h)}, \ldots, X_{i+\ell}^{(h)}$ on height h such that $\pi^*(C)$ is a substring of $\pi^*(X_i^{(h)} \cdots X_{i+\ell}^{(h)})$ such that $\pi^*(C)$ is a suffix of $\pi^*(X_i^{(h)})$ and a prefix of $\pi^*(X_{i+\ell}^{(h)})$. Given there are non-terminals $X_j^{(h+1)}$ and $X_{j'}^{(h+1)}$ being the parents of $X_i^{(h)}$ and $X_{i+\ell}^{(h)}$, respectively, with $j \neq j'$, then we cannot extend $\pi^*(X_i^{(h)} \cdots X_{i+\ell}^{(h)})$ to an occurrence of P in T.

A consequence of the latter property is that for each occurrence O of C in $\mathcal{T}_T$ whose expansion is contained in an occurrence of P, this occurrence O is a (not necessarily proper) substring of the RHS of a rule of $\mathcal{G}_T$.

Our definition of *core* is neither constructive nor unique. In fact, there can be many cores for a given pattern. We qualify a core by the difference in the number of occurrences of P and C in $\mathcal{T}_T$. On the one hand, although a character $P[i]$ always qualifies as a core, the appearance of $P[i]$ in T is unlikely to be evidence for an occurrence of P. (Additionally, our algorithm would perform worse than standard pattern matching with the suffix tree when resorting to single characters as cores). On the other hand, the non-terminal covering most of the characters of P might not be a core. Hence, we aim for the highest possible non-terminal, for which we are sure that it exhibits the core property.

In what follows, we present a greedy heuristic that finds a core C of P during the construction of the GCIS grammar $\mathcal{G}_P$ of P. The idea is to discard sufficiently long prefixes and suffixes on each recursion step of the GCIS construction of P, allowing us to guarantee the core properties for the remaining center part. Since we shave off bordering symbols at each height, the remaining part has the shape of a pyramid, where the topmost (and inmost) non-terminal is our core C.

Finding a Core We determine a core C of P during the computation of the GCIS grammar $\mathcal{G}_P$ of P. During this computation, existing text non-terminals are reused when lookup succeeds, i.e., if $\mathsf{lookup}(F) = X$ for a factor F, we reuse the non-terminal X from $\mathcal{G}_T$ instead of creating a new non-terminal for F in $\mathcal{G}_P$. By doing so, we ensure that non-terminals of $\mathcal{G}_P$ and $\mathcal{G}_T$ are identical whenever their RHSs are equal. In detail, suppose that we are on height h and parse $P^{(h)}$ into LMS factors $F_1^{(h)} \cdots F_{z_h}^{(h)} = P^{(h)}$. Next, we retrieve $Y_i^{(h+1)} := \mathsf{lookup}(F_i^{(h)})$ for each $i \in [2..z_h - 1]$, effectively omitting the first and last factor of the factorization of $P^{(h)}$. If one of the lookup-queries returns $\bot$, we abort since we can be sure that the pattern does not occur in T. That is because all non-terminals $Y_2^{(h+1)}, \ldots Y_{z_h-1}^{(h+1)}$ are cores. To see this, we observe that prepending or appending symbols to $P^{(h)}$ does not change the factors $F_2^{(h)}, \ldots, F_{z-1}^{(h)} =: C^{(h)}$. We give a visual interpretation with Fig. 1.

Correctness We show that prepending or appending characters to $F_1^{(h)}\ C^{(h)}\ F_{z_h}^{(h)}$ does not modify the computed factorization of $C^{(h)} = F_2^{(h)}, \ldots, F_{z-1}^{(h)}$. For that, we observe that we cannot change the type of any position $C^{(h)}[i]$ to $\mathtt{S}^*$ by prepending or appending characters.

Prepending Firstly, the type of a position (`S` or `L`) depends only on its succeeding position, and hence prepending cannot change the type of a position in $C^{(h)}$.

Appending Secondly, appending characters can either prolong $F_{z_h}^{(h)}$ or create a new factor $F_{z_{h+1}}^{(h)}$ since $F_{z_h}^{(h)}$ starts with $\mathtt{S}^*$, and therefore appending cannot change $C^{(h)}$.

An additional insight is that on the one hand, prepending a symbol can only introduce a new factor or extend $F_1^{(h)}$. On the other hand, appending characters can introduce at most one new $\mathtt{S}^*$ position in $F_{z_h}^{(h)}$ that can make it split into two factors. We will need this observation later for extending the core to the pattern, cf. Fig. 4.

The construction of $\mathcal{G}_P$ iterates the LMS factorization until we are left with a string of symbols $P^{(\tau_P)}$ whose LMS factorization consists of at most two factors. In that case, we partition $P^{(\tau_P)}$ into three substrings $C_p \cdot C \cdot C_s$ with C_p and C_s possibly empty, and defined by the two independent rules for C_p and C_s.

For C_p: If the LMS factorization consists of two non-empty factors $F_1 \cdot F_2$, then C_p is F_1.

For C_s: Let $P^{(\tau_P)} = (P^{(\tau_P)}[j_1])^{c_1} \cdots (P^{(\tau_P)}[j_k])^{c_k}$ be the run-length-encoded representation of $P^{(\tau_P)}$ with $1 = j_1 < \ldots < j_k = |P^{(\tau_P)}|$, $c_i \geq 1$ for $i \in [1..k]$, and $P^{(\tau_P)}[j_i] \neq P^{(\tau_P)}[j_{i+1}]$ for $i \in [1..k-1]$. We set $C_s \leftarrow (P^{(\tau_P)}[j_k])^{c_k}$ if $P^{(\tau_P)}[j_k] < P^{(\tau_P)}[j_{k-1}]$.

In the other cases, C_p and/or C_s are empty. Figure 2 visualizes C_p and C_s based on the established rules. While we cannot guarantee that C_p and C_s are cores of P, we can show the following.

Lemma 4.1. *C is a core of P.*

Proof. It is left to check the case when C_s is empty. The other cases have already been covered by the aforementioned analysis of the cores on the lower heights. Let y denote the last symbol of $P^{(\tau_P)}$. If C_s is empty, then C is a suffix of P, and as a border case, the last position of C is $\mathtt{S}^*$. In that case, appending symbols cannot create an LMS boundary inside C: It can change the type of the last position of C, but the affected boundary is after the last symbol of C.

- If we append a string $y^\ell x$ with $x < y$ and $\ell \geq 0$ to C_pC, then the type of the last position of C changes to `L`.
- If we append a string $y^\ell z$ with $z > y$ and $\ell \geq 0$, then the last position of C becomes `S`, but does not become $\mathtt{S}^*$ since $P^{(\tau_P)}[j_k] > P^{(\tau_P)}[j_{k-1}]$ due to construction (otherwise C_s would not be empty).

□

In particular, if C_p or C_s is empty, then the expansion of the core C is a prefix or a suffix of the pattern, respectively.

We can also extract more cores, but for these we can only guarantee that they have a lower height than C. To put in technical terms, there are symbols $A^{(1)}, \ldots, A^{(\tau_P-2)}$ and $S^{(1)}, \ldots, S^{(\tau_P-2)}$ such that

$$\begin{aligned} D_p &:= F_1^{(1)} \cdots F_1^{(\tau_P-2)} A^{(1)} \cdots A^{(\tau_P-2)} \\ D_s &:= S^{(\tau_P-2)} \cdots S^{(1)} F_{z_{\tau_P-2}}^{(\tau_P-2)} \cdots F_{z_1}^{(1)} \\ P &= \pi^*(D_p C D_s) \text{ with } \pi^*(D_p) = \pi^*(C_p) \text{ and } \pi^*(D_s) = \pi^*(C_s), \end{aligned} \tag{1}$$

and $A^{(h)}$, $S^{(h)} \in \Gamma^{(h)}$ are cores of P, while $F_1^{(h)}$, $F_{z_h}^{(h)} \in (\Gamma^{(h-1)})^*$ are factors, for each height $h \in [1..\tau_P - 2]$.

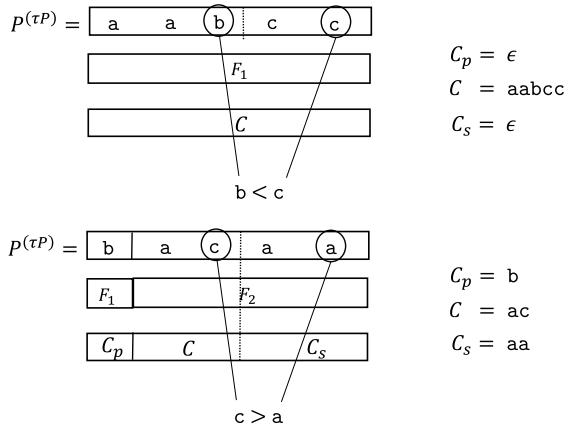

Figure 2: Determining the core C of P from $P^{(\tau_P)}$. Top: The LMS factorization of $P^{(\tau_P)}$ consists only of one factor F_1, so C_p is empty. The last character run `cc` consists of a character larger than the previous run, which consists of `b`'s, such that C_s is empty, too. Bottom: The factorization consists of two factors F_1F_2, so $C_p = F_1$. Since the last character run `aa` is lexicographically smaller than the previous one with `c`, C_s is set to this last character run.

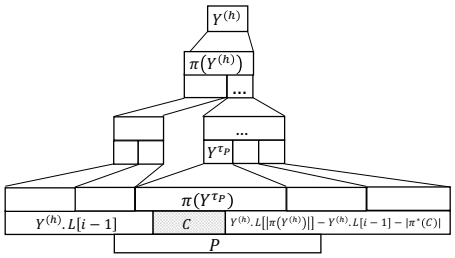

Figure 3: Deriving a non-terminal $Y^{(h)}$ with $\pi^*(Y^{(h)})$ containing P from a non-terminal $Y^{(\tau_P)}$ with $\pi^*(Y^{(\tau_P)})$ containing $\pi^*(C)$. The expansion of none of the descendants of $Y^{(h)}$ towards $Y^{(\tau_P)}$ is large enough for extending its contained occurrence of $\pi^*(C)$ to an occurrence of P. We can check the expansion lengths of the substrings in $\pi(Y^{(h)})$ via the array $Y^{(h)}.L$.

4.2 Matching with GST

Having C, we now switch to GST and use it to find all DAG parents of C, whose number we denote by $\text{occ}_C \in \mathcal{O}(g)$. This number is also the number of occurrences of C in the RHSs of all rules of $\mathcal{G}_T$. Having these parents, we want to find all lowest DAG ancestors of C whose expansions are large enough to not only cover C but also P by extending C to its left and right side — see Fig. 3 for a sketch. We proceed as follows: We first compute the locus v of C in GST in $\mathcal{O}(|C| \lg |\mathcal{S}|)$ time via child. Subsequently, we take the pointer to the leftmost leaf in the subtree rooted at v, and then process all leaves in this subtree by using the linked list of leaves. For each such leaf λ, we compute a path in form of a list λ_L from the non-terminal containing C on its RHS up to an ancestor of it that has an expansion large enough to cover P if we would expand the contained occurrence of C to P. We do so as follows: Each of these leaves stores a pointer to a non-terminal X and a starting position i such that we know that $\pi^*(X)[i..]$ starts with $\pi^*(C)$. By knowing the expansion lengths $X.L[|\pi(X)|]$, $X.L[i-1]$, and $|\pi^*(C)|$, we can judge whether the expansion of X has enough characters to be able to extend its occurrence of C to P. If it has enough characters, we put (X, i) onto λ_L such that we know that $\pi^*(X)[X.L[i-1]+1..]$ has C as a prefix. If X does not have enough characters, we exchange C with X and recurse on finding a non-terminal with a larger expansion. By doing so, we visit at most $\tau_T = \mathcal{O}(\lg n)$ non-terminals per occurrence of C in the RHSs of $\mathcal{G}_T$. We perform all operations in $\mathcal{O}(\text{occ}_C \tau_T \lg |\mathcal{S}|)$ time because we query child in every recursion step.

Grammar Subtrees Containing P The previous step computes, for each accessed leaf λ, a list λ_L containing a DAG path $(Y^{(h^*)}, \ldots, Y^{(\tau_P)})$ of length $\mathcal{O}(\tau_T)$ and an offset $o^{(\tau_P)}$ such that

- $|P| \leq |\pi^*(Y^{(h^*)})|$,
- $Y^{(h^*)}$ is an ancestor of $Y^{(\tau_P)}$, and
- $Y^{(\tau_P)}[o^{(\tau_P)}..]$ starts with C.

By construction, these paths cover all occurrences of C in $\mathcal{T}_T$. We process the DAG node $Y^{(\tau_P)}$ (with different offsets $o^{(\tau_P)}$) as many times as C occurs in $\pi(Y^{(\tau_P)})$. In what follows, we try to

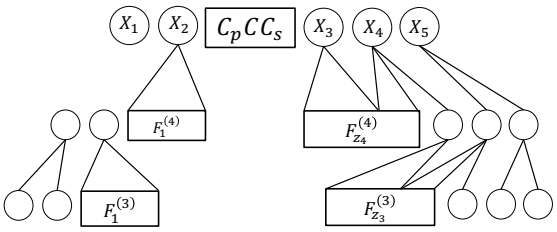

Figure 4: Occurrence of C in $\mathcal{T}_T$. Circles symbolize nodes, rectangles multiple sibling nodes. To check whether this occurrence corresponds to an occurrence of P, it suffices to check the extensions of $X_1X_2C_pCC_sX_3X_4X_5$. Although X_1, X_2, X_3, X_4, X_5 are not cores, $\pi(X_2)$, $\pi(X_3)$ and $\pi(X_4)$ extend to symbols that match with symbols in the derivation tree of $\mathcal{G}_P$. For the pattern matching, it suffices to match $F_1^{(h)}$ and two symbols to its left, and $F_{z_h}^{(h)}$ and three symbols to its right (cf. Sect. 4.1).

expand the occurrence of C captured by $Y^{(\tau_P)}$ and $o^{(\tau_P)}$ to an occurrence of P. See Fig. 4 for a sketch of the setting. Naively, we would walk down from $Y^{(\tau_P)}[o^{(\tau_P)}]$ to the character level and extend the substring $\pi^*(C)$ in both directions by character-wise comparison with P. However, this would take $\mathcal{O}(\mathrm{occ}_C m\tau_T)$ time since such a non-terminal $Y^{(h^*)}$ is of height $\mathcal{O}(\tau_T)$. Our claim is that we can perform the computation in $\mathcal{O}(m + \mathrm{occ}_C\tau_T)$ time with the aid of lce and an amortization argument.

For that, we use Eq. (1), which allows us to use LCE queries in the sense that we can try to extend an occurrence of C with an already extended occurrence (that may not match P completely) from a different leaf λ and hence a possibly different path (such an extension is safe since we retrieve a substring of consecutive cores). For the explanation, we only focus on extending all occurrences of C to the right to CC_s (the left side is done symmetrically), cf. Fig. 5. We maintain an array D of length $\tau_P - 1$ storing pairs $(X^{(h)}, \ell_h)$ with $X^{(h)} \in \Gamma^{(h)}$ and $\ell_h \geq 0$ for each height $h \in [1..\tau_P - 1]$ such that $\pi(X^{(h)})$ has the currently longest extension of length ℓ_h with the core $S^{(h-1)}$ of P in common (cf. Eq. (1)). By maintaining D, we can first query lce with the specific non-terminal in D, and then resort to plain symbol comparison. We descend to the child where the mismatch happens and recurse until reaching the character level of $\mathcal{T}_T$. This all works since by the core property the mismatch of a child means that there is a mismatch in the expansion of this child. Since a plain symbol comparison with matching symbols lets us exchange the currently used non-terminal in D with a longer matched prefix or a different non-terminal with larger expansion, we can bound (a) the total number of naive symbol matches to $\mathcal{O}(m)$ and (b) the total number of naive symbol mismatches and LCE queries to $\mathcal{O}(\mathrm{occ}_C\tau_T)$. This concludes the right extension of C. By symmetry, we also perform LCE queries in reversed direction to match the cores $A^{(h)}$ on the left side of C in Eq. (1) for all $h \in [1..\tau_P - 2]$ to extend C to the left.

5 LCE Queries in Detail

Put in technical terms, we start with D storing only invalid entries $\perp$ to indicate that there is no candidate on any height for LCE queries. Suppose that we are given a DAG path $\mathsf{path} := (Y^{(h^*)}, \ldots, Y^{(\tau_P)})$ to an occurrence of C and an offset $o^{(\tau_P)}$ such that $\pi(Y^{(\tau_P)})[o^{(\tau_P)}..]$ starts with C.

In a procedure called NextRightSymbol in Algorithm 2, we obtain the highest-positioned symbol immediately to the right of the occurrence of C along the given DAG path. This skips levels where the expansion of C is a suffix of the current non-terminal expansion. If the required symbol does not exist, the occurrence is rejected; if the right extension is already complete, we proceed directly to the final verification step. Given this symbol is at height j_0, we scan the entries of D from $h = j_0$ to $h = 1$ while checking whether $D[h] = \perp$:

- Suppose that $D[h] = (\tilde{Y}^{(h)}, \tilde{\ell}^{(h)})$ is not empty, we compute $\ell := \mathsf{lce}(\tilde{Y}^{(h)}, 1, Y^{(h)}, 1)$. We abort if $\ell < \tilde{\ell}^{(h)}$. Otherwise, we update $D[h] := (Y^{(h)}, \ell)$, set $Y^{(h-1)} := \pi(Y^{(h)})[\tilde{\ell}^{(h)} + 1]$, and recurse.

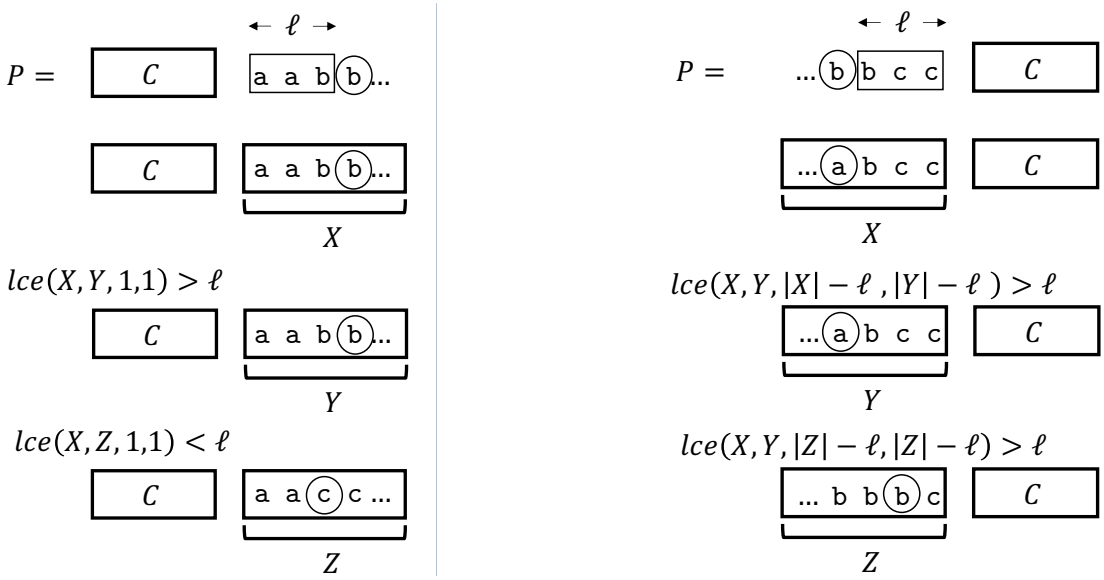

Figure 5: LCE queries issued in Sect. 4.2. Left: Let $X, Y, Z \in \Gamma^{(h)}$. Assume that the expansion of the first ℓ symbols of the non-terminal X coincides with the expansion of C_s, and that X is currently stored in D. To compute the number of matching characters of the expansion of C_s with the expansion of Y or Z, we first issue LCE queries with X to decide whether there is a mismatch at one of the first ℓ symbols; in such a case we know that we cannot find an occurrence of the pattern, as it is the case with Z. Right: Symmetric application for extending C to the left. The LCE queries compare symbols in RHSs of the non-terminals.

- If $D[h] = \bot$, we compare $\pi(Y^{(h)})$ with the respective cores of the pattern at height $h-1$, cf. Eq. (1) and Fig. 4. Given we matched ℓ symbol pairs, we set $D[h] := (Y^{(h)}, \ell)$.

On reaching $h = 1$, we compare the RHSs of $Y^{(1)}$ and its right siblings with the remaining part of the pattern. On reaching a mismatch or completing $\pi^*(C_s)$, we update the entries of D while climbing up path, and store the node $Y^{(h^*)}$ with the starting position of the respective occurrence of P relative to its grammar subtree in a list W. The traversal, the update of D, and LCE queries take $\mathcal{O}(\tau_T)$ time per occurrence of C, while the total number of naive symbol matches accumulate to $\mathcal{O}(m)$ time. We need the path of λ_L to start at $Y^{(h^*)}$, since otherwise selecting sibling nodes on the same height can cost $\mathcal{O}(\tau_T)$ time. Algorithm 1 and Algorithm 2 show pseudocode for the locate procedure described above. The algorithm consists of several key components: computing the core, extending occurrences using LCE queries, and finding the starting positions, which we describe next.

Finding the Starting Positions It is left to compute the starting position in T of each occurrence captured by an element in W. We can do this similarly to computing the pre-order ranks in a tree: For each pair $(X, \ell) \in W$, climb up DAG from X to the root while accumulating the expansion lengths of all left siblings of the nodes we visit (we can make use of $Y.L$ for Y being a parent of X). If this accumulated length is s, then $\ell + s$ is the starting position of the occurrence captured by (X, ℓ). However, this approach would cost $\mathcal{O}(\tau_T)$ time per element of W. Here, we use the amortization argument of Claude and Navarro [7, Sect. 5.2], which works if we augment, in a pre-computation step, each non-terminal X in Γ with (a) a pointer to the lowest ancestor Y_X on every path from X to the DAG root that has X at least twice as a descendant, and (b) the lengths of the expansions of the left siblings of the child of Y_X being a parent of X or X itself. By doing so, when taking a pointer of a non-terminal X to its ancestor Y_X, we know that X has another occurrence in DAG (and thus there is another occurrence of P). Therefore, we can charge the cost

Algorithm 1 Locate: Finding all occurrences of P in T.

Require: Pattern P of length m, Text T with grammar $\mathcal{G}_T$
Ensure: List of starting positions of all occurrences of P in T

Step 1: Compute grammar $\mathcal{G}_P$ of pattern P and determine core C

Initialize: $h \leftarrow 0$, $P^{(0)} \leftarrow P$
while LMS factorization of $P^{(h)}$ has > 2 factors **do**
  Compute LMS factorization: $P^{(h)} = F_1^{(h)} \cdots F_{z_h}^{(h)}$
  for $i = 2$ **to** $z_h - 1$ **do**
    $X_i^{(h+1)} \leftarrow \mathsf{lookup}(F_i^{(h)})$
    if $X_i^{(h+1)} = \perp$ **then return** $\emptyset$ ▷ *pattern not in text*
  $P^{(h+1)} \leftarrow X_2^{(h+1)} \cdots X_{z_h-1}^{(h+1)}$
  $h \leftarrow h + 1$
$\tau_P \leftarrow h$ ▷ *Height of derivation tree of pattern grammar* $\mathcal{G}_P$
Determine core C from $P^{(\tau_P)}$ using rules in Sect. 4.1

Step 2: Find all occurrences of core C in GST

Compute locus v of C in GST using child queries
$L \leftarrow \emptyset$
for each leaf λ in subtree rooted at v **do**
  Extract $(Y^{(\tau_P)}, o^{(\tau_P)})$ from λ
  for each DAG path $(Y^{(h^*)}, \ldots, Y^{(\tau_P)})$ to lowest ancestor $Y^{(h^*)}$ with $|P| \leq |\pi^*(Y^{(h^*)})|$ **do**
    Add $((Y^{(h^*)}, \ldots, Y^{(\tau_P)}), o^{(\tau_P)})$ to list L ▷ *Enumeration is over parent occurrences and can be implemented with a stack, cf. Step 4.*

Step 3: Extend core occurrences to pattern occurrences (if possible)

Initialize $D[1..\tau_P - 1] \leftarrow [\perp, \ldots, \perp]$ ▷ *Gets filled in* ExtendOccurrence
Initialize $W \leftarrow \emptyset$
for each $((Y^{(h^*)}, \ldots, Y^{(\tau_P)}), o^{(\tau_P)})$ in L **do**
  ExtendOccurrence$((Y^{(h^*)}, \ldots, Y^{(\tau_P)}), o^{(\tau_P)}, D, W)$ ▷ *cf. Algorithm 2*

Step 4: Compute starting positions in T

Initialize result $\leftarrow \emptyset$
for each (X, ℓ) in W **do**
  Initialize visit stack $\mathcal{V} \leftarrow [(X, \ell)]$
  while $\mathcal{V}$ is not empty **do**
    $(\text{curr}, \text{pos}) \leftarrow \text{POP}(\mathcal{V})$
    if curr is the root of DAG **then**
      Add pos to result
    else
      for each parent occurrence (Y, i) of curr in DAG **do** ▷ $\pi(Y)[i] = curr$
        $\text{pos}' \leftarrow \text{pos} + Y.L[i-1]$ ▷ $Y.L[i-1] = \sum_{j=1}^{i-1} |\pi^*(\pi(Y)[j])|$ *and* $Y.L[0] = 0$
        PUSH$(\mathcal{V}, (Y, \text{pos}'))$
return result

of climbing up the tree with the amount of occurrences occ of the pattern.

Total Time To sum up, we need $\mathcal{O}(m \lg |\mathcal{S}|)$ time for finding C, $\mathcal{O}(\text{occ}_C \tau_T \lg |\mathcal{S}|)$ time for computing the non-terminals covering C, $\mathcal{O}(m + \text{occ}_C \tau_T)$ time for reducing these non-terminals to W, and $\mathcal{O}(\text{occ})$ time for retrieving the starting positions of the occurrences of P in T from W. To be within our $\mathcal{O}(g)$ space bounds, we can process each DAG parent of C individually, and keep only D globally stored during the whole process. The total additional space is therefore $\mathcal{O}(\tau_T) \subset \mathcal{O}(g)$ for maintaining D and a path for each occurrence of C. So we finally obtain the claim of Thm. 1.1.

Algorithm 2 Extend a core occurrence to check if it corresponds to a pattern occurrence. NextRightSymbol returns the highest-positioned symbol immediately to the right of the occurrence of C along the given DAG path, skipping levels where the expansion C is a suffix of the expansion of the respective non-terminal; it returns $\perp$ if the right extension is already complete; if the required symbol is not available, the occurrence is rejected.

Require: DAG path $(Y^{(h^*)}, \ldots, Y^{(\tau_P)})$, offset $o^{(\tau_P)}$, array D, result list W
Ensure: Updates W with valid pattern occurrences

$(j_0, Y^{(j_0)}) \leftarrow \text{NextRightSymbol}((Y^{(h^*)}, \ldots, Y^{(\tau_P)}), o^{(\tau_P)}, C)$
if $(j_0, Y^{(j_0)}) = \perp$ **then**
    goto FinalVerification
for $j = j_0$ **down to** 1 **do**
    if $D[j] \neq \perp$ **then**
        Let $D[j] = (\tilde{Y}^{(j)}, \tilde{\ell}^{(j)})$ ▷ $\tilde{\ell}^{(j)}$ *is the length of the longest common prefix so far*
        $\ell \leftarrow \mathsf{lce}(\tilde{Y}^{(j)}, 1, Y^{(j)}, 1)$
        if $\ell < \tilde{\ell}^{(j)}$ **then return** ▷ *abort this occurrence*
        else
            $Y^{(j-1)} \leftarrow \pi(Y^{(j)})[\tilde{\ell}^{(j)} + 1]$
            continue to next iteration
    else if $D[j] = \perp$ **then**
        $\ell^{(j)} \leftarrow$ longest common prefix of $\pi(Y^{(j)})$ and pattern cores at height $j - 1$
        Update $D[j] \leftarrow (Y^{(j)}, \ell^{(j)})$
        $Y^{(j-1)} \leftarrow \pi(Y^{(j)})[\ell^{(j)} + 1]$
        continue to next iteration
FinalVerification:
Compare $\pi(Y^{(1)})$ and right siblings with remaining pattern
if complete match found **then**
    Compute starting position ℓ relative to $Y^{(h^*)}$ from the input DAG path
    Add $(Y^{(h^*)}, \ell)$ to W
Update D while climbing back up the path

6 Walk-through Example

To illustrate the GCIS index construction and pattern matching algorithm, we present a detailed example using a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ with $\mathtt{a} < \mathtt{b}$.

Text and Pattern Consider the text $T = \mathtt{abaababaabaab}$ of length $n = 13$ and pattern $P = \mathtt{abaab}$ of length $m = 5$. The pattern occurs 3 times in the text at positions 1, 6, and 9.

Building the GCIS Grammar of the Text We first construct the GCIS grammar $\mathcal{G}_T$ for the text T.

Height 0: $T^{(0)} = \mathtt{abaababaabaab}$
We identify the suffix types:

- Positions with type S: $\{1, 3, 4, 6, 8, 9, 11, 12, 13\}$
- Positions with type L: $\{2, 5, 7, 10\}$
- Positions with type S* (LMS): $\{1, 3, 6, 8, 11\}$

The LMS factorization gives us:

$$\#T = \#\mathtt{ab} \cdot \mathtt{aab} \cdot \mathtt{ab} \cdot \mathtt{aab} \cdot \mathtt{aab}$$

After omitting the delimiter #, we obtain factors:

$$F_1^{(0)} = \texttt{ab}$$
$$F_2^{(0)} = \texttt{aab}$$
$$F_3^{(0)} = \texttt{ab}$$
$$F_4^{(0)} = \texttt{aab}$$
$$F_5^{(0)} = \texttt{aab}$$

We assign non-terminals $X_1^{(1)} \to \texttt{ab}$ and $X_2^{(1)} \to \texttt{aab}$ with $X_1^{(1)} \succ X_2^{(1)}$ based on the lexicographic order of their expansions.

Height 1: $T^{(1)} = X_1^{(1)} X_2^{(1)} X_1^{(1)} X_2^{(1)} X_2^{(1)}$

The LMS factorization yields:

$$T^{(1)} = (X_1^{(1)}) \cdot (X_2^{(1)} X_1^{(1)} X_2^{(1)} X_2^{(1)})$$

We create: $X_1^{(2)} \to X_1^{(1)}$ and $X_2^{(2)} \to X_2^{(1)} X_1^{(1)} X_2^{(1)} X_2^{(1)}$

Thus: $T^{(2)} = X_1^{(2)} X_2^{(2)}$

Since the factorization has only 2 factors, we stop here with $\tau_T = 2$ and start symbol $X^{(3)} \to X_1^{(2)} X_2^{(2)}$.

The complete grammar $\mathcal{G}_T$ consists of:

$$X^{(3)} \to X_1^{(2)} X_2^{(2)}$$
$$X_1^{(2)} \to X_1^{(1)}$$
$$X_2^{(2)} \to X_2^{(1)} X_1^{(1)} X_2^{(1)} X_2^{(1)}$$
$$X_1^{(1)} \to \texttt{ab}$$
$$X_2^{(1)} \to \texttt{aab}$$

with grammar size $g = 2 + 1 + 4 + 2 + 3 = 12$.

Building the GCIS Grammar of the Pattern Now we construct $\mathcal{G}_P$ for pattern $P = \texttt{abaab}$.

Height 0: $P^{(0)} = \texttt{abaab}$

LMS positions are $\{1, 3\}$, giving factorization:

$$\#P = \#\texttt{ab} \cdot \texttt{aab}$$

We check $\mathsf{lookup}(\texttt{ab}) = X_1^{(1)}$ and $\mathsf{lookup}(\texttt{aab}) = X_2^{(1)}$. Both exist in $\mathcal{G}_T$. Thus, we have $P^{(1)} = X_1^{(1)} X_2^{(1)}$. Since we are left with two symbols, we already stop here with $C_p = X_1^{(1)}$, $C = X_2^{(1)}$.

Finding and Extending Core Occurrences C appears three times in the RHS of $X_2^{(2)}$, so the number of core occurrences occ_C is three. For each core occurrence, we check if it can be extended to the full pattern $P = \pi^*(X_1^{(1)} X_2^{(1)})$: The second and third occurrence of C in $X_2^{(2)}$ have a sufficient distance from the beginning of the expansion of $X_2^{(2)}$ to accommodate the full pattern, so we perform reversed-LCE queries on them to find the longest common suffix with $C_p = \texttt{ab}$. We thus obtain the second occurrence of P at position 6 and the third occurrence of P at position 9. For the first occurrence, we need to check each parent of $X_2^{(2)}$ to see if it can accommodate the full pattern. There is only one parent $X^{(3)}$ of $X_2^{(2)}$, and the left sibling of $X_2^{(2)}$ in $X^{(3)}$ is $X_1^{(2)}$, on which we apply reversed-LCE queries to check if it can match C_p. This gives us the first occurrence of P at position 1 in the expansion of $X^{(3)}$.

7 Implementation

The implementation deviates from theory with respect to the rather large hidden constant factor in the $\mathcal{O}(g)$ words of space. We drop GST, and represent DAG with multiple arrays. For that,

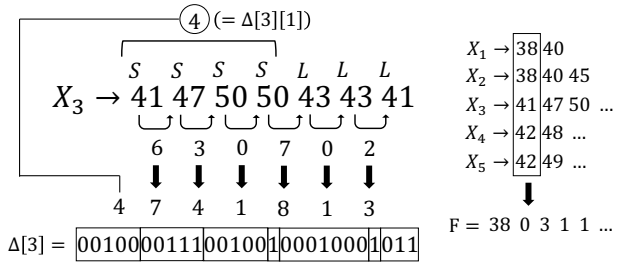

Figure 6: Encoding of GCIS-uni. *Right:* Encoding the first symbol of each RHS with Elias–Fano and storing the sequence in an integer array F. *Left:* Storing the remaining characters of $\pi(X_3)$ in $\Delta[3]$ delta-encoded with Elias-γ. $\pi(X_3)$ is a bitonic sequence, where the first $\Delta[3][1] = 4$ symbols are increasing. The bottom row depicts the binary representation of $\Delta[3]$.

we first enumerate the non-terminals as follows: The height and the lexicographic order induce a natural order on the non-terminals in Γ, which are ranked by first their height and secondly by the lexicographic order of their RHSs, such that we can represent $\Gamma = \{X_1, \ldots, X_{|\Gamma|}\}$. By stipulating that all characters are lexicographically smaller than all non-terminals, we obtain the property that $\pi(X_i) \prec \pi(X_{i+1})$ for all $i \in [1..|\Gamma| - 1]$. In the following, we first present a plain representation of DAG, called GCIS-nep, then give our modified locate algorithm, and subsequently present a compressed version of DAG using universal coding, called GCIS-uni. Finally, we evaluate both implementations in Sect. 8.

Our first implementation, called GCIS-nep[2], represents each symbol with a 32-bit integer. We use $R := \prod_{i=1}^{|\Gamma|} \pi(X_i)$ again, but omit the delimiters \$ separating the RHSs. To find the RHS of a non-terminal X_i, we create an array of positions $Q[1..|\Gamma|]$ such that $Q[i]$ points to the starting position of $\pi(X_i)$ in R. Finally, we create an array $L[1..|\Gamma|]$ storing the length of the expansion $|\pi^*(X_i)|$ in $L[i]$, for each non-terminal X_i. Due to the stipulated order of the symbols, the strings $R[Q[i]..Q[i+1]-1]$ are sorted in ascending order. Hence, we can evaluate lookup(S) for a string S in $\mathcal{O}(|S| \lg |\mathcal{S}|)$ time by a binary search on Q with $i \mapsto R[Q[i]..Q[i+1]-1]$ as keys.

Locate Our implementation follows theory for computing $\mathcal{G}_P$ and C (cf. Sect. 4.1) in the same time bounds, but deviates after computing the core C: To find all non-terminals whose RHSs contain C, we linearly scan the RHSs of all non-terminals on height τ_P, which we can do cache-friendly since the RHSs of R are sorted by the height of their respective non-terminals. This takes $\mathcal{O}(g + |C|)$ time in total with a pattern matching algorithm [30]. Finally, for extending a found occurrence of the core C to an occurrence of P, we follow the naive approach to descend DAG to the character level and compare the expansion with P character-wise, which results in $\mathcal{O}(\mathrm{occ}_C |P| \tau_T)$ time. The total time cost is $\mathcal{O}(g + |P|(\mathrm{occ}_C \tau_T + \lg |\mathcal{S}|))$.

GCIS-uni To save space, we can leverage universal code to compress the RHSs of the productions. First, we observe that Q and the first symbols $F := \pi(X_1)[1], \ldots, \pi(X_{|\Gamma|})[1]$ form an ascending sequence, such that we represent both Q and F in Elias–Fano coding [16]. Next, we observe that each RHS $\pi(X_i)$ forms a bitonic sequence: the ranks of the first ℓ_i symbols are non-decreasing, while the rest of the ranks are non-increasing. Our idea is to store ℓ_i and the rest of $\pi(X_i)[2..]$ in delta-coding, i.e., $\Delta[i][k] := |\pi(X_i)[k] - \pi(X_i)[k-1]|$ for $k \in [2..|\pi(X_i)|]$, which is stored in Elias-γ code [17]. Although $\pi(X_i)[k] - \pi(X_i)[k-1] < 0$ for $k > \ell_i$, we can decode $\pi(X_i)[k]$ by subtracting instead of adding the difference to $\pi(X_i)[k-1]$ as usual in delta-coding. Hence, we can replace R with Δ, but need to adjust Q such that $Q[i]$ points to the first bit of $\Delta[i]$. Finally, like in the first variant, we store the expansion lengths of all non-terminals in L. Here, we separate L in a first part using 8 bits per entry, then 16 bits per entry, and finally 32 bits per entry. To this end, we represent L by three arrays, start with filling the first array, and continue with filling the next array whenever we process a value whose bit representation cannot be stored in a single entry of the current array. Since Elias–Fano code supports constant-time random access and Elias-γ supports constant time

[2]GCIS-nep stands for GCIS with **n**on-terminals **e**ncoded **p**lainly.

Table 3: Top: Sizes of the used datasets and the indexes stored on disk. Sizes are in megabytes [MB]. Bottom: Statistics of the GCIS grammar.

dataset	input size	GCIS-nep	GCIS-uni	ESP-index	FM-index	r-index
COMMONCRAWL	221.180	220.119	138.856	156.006	122.575	454.124
DNA	403.927	527.553	327.852	297.001	216.153	2123.817
EINSTEIN.DE	92.758	1.139	0.428	0.697	40.291	1.146
ENGLISH.001.2	104.857	14.784	7.489	10.464	46.981	14.389
FIB41	267.914	0.001	0.001	0.001	71.305	0.007
INFLUENZA	154.808	23.373	13.871	15.729	53.066	28.775
KERNEL	257.961	21.298	10.469	12.545	125.087	28.947
RS.13	216.747	0.002	0.001	0.002	57.653	0.009
TM29	268.435	0.002	0.001	0.002	69.347	0.009
WORLD LEADERS	46.968	5.415	2.573	3.611	21.097	5.627

dataset	$\lvert\Gamma\rvert$	g	$\lvert\pi(X^{(\tau_T)})\rvert$
COMMONCRAWL	8565089	37899804	7603482
DNA	21068903	89750482	11462490
EINSTEIN.DE	50554	183664	3491
ENGLISH.001.2	661175	2373818	152854
FIB41	67	173	22
INFLUENZA	971626	3900153	477072
KERNEL	943991	3436501	2478
TM29	104	311	16
WORLD LEADERS	246589	860611	31743

per decoded codeword during sequential access, we can decode $\pi(X_i)$ by accessing $F[i]$ and then sequentially decode $\Delta[i]$. Hence, we can simulate GCIS-nep with this compressed version without sacrificing the theoretical bounds. We call the resulting index GCIS-uni. Figure 6 captures our compression steps.

8 Experiments

In the following we present an evaluation of our C++ implementation and different self-indexes for comparison, which are the FM-index [20], the ESP-Index [41], and the r-index [23][3]. All code has been compiled with `gcc-10.2.0` in the highest optimization mode `-O3`. We ran all our experiments on a Mac Pro Server with an Intel Xeon CPU X5670 clocked at 2.93GHz running Arch Linux.

Our datasets shown in Table 3 are from the Pizza&Chili and the tudocomp [14] corpus.[4] With respect to the index sizes, we empirically observe the ranking GCIS-uni < ESP-index < GCIS-nep, followed by one of the BWT-based indexes. While the r-index needs less space than the FM-index on highly-compressible datasets, it is the least favorable option of all indexes for less-compressible datasets. Figure 7 gives the time and memory needed for constructing the indexes. There, we measured the memory as the *maximum resident set size* reported by `/usr/bin/time`.

The bottom of Table 3 presents some characteristics of the GCIS grammar, where $|\pi(X^{(\tau_T)})|$ is the size of the start symbol. We can see that the number of non-terminals, grammar size g, and the size of the start symbol scale with the compressibility of the input. Especially artificially generated datasets compress well with GCIS.

We can observe in Fig. 8 that our indexes answer locate(P) fast when P is sufficiently long or has many occurrences occ in T. GCIS-uni is always slower than GCIS-nep due to the extra costs

[3]See `https://github.com/mpetri/FM-Index`, `https://github.com/tkbtkysms/esp-index-I`, and `https://github.com/nicolaprezza/r-index`, respectively.

[4]To save space, we renamed the datasets COMMONCRAWL.ASCII.TXT and EINSTEIN.DE.TXT to COMMONCRAWL and EINSTEIN.DE, respectively.

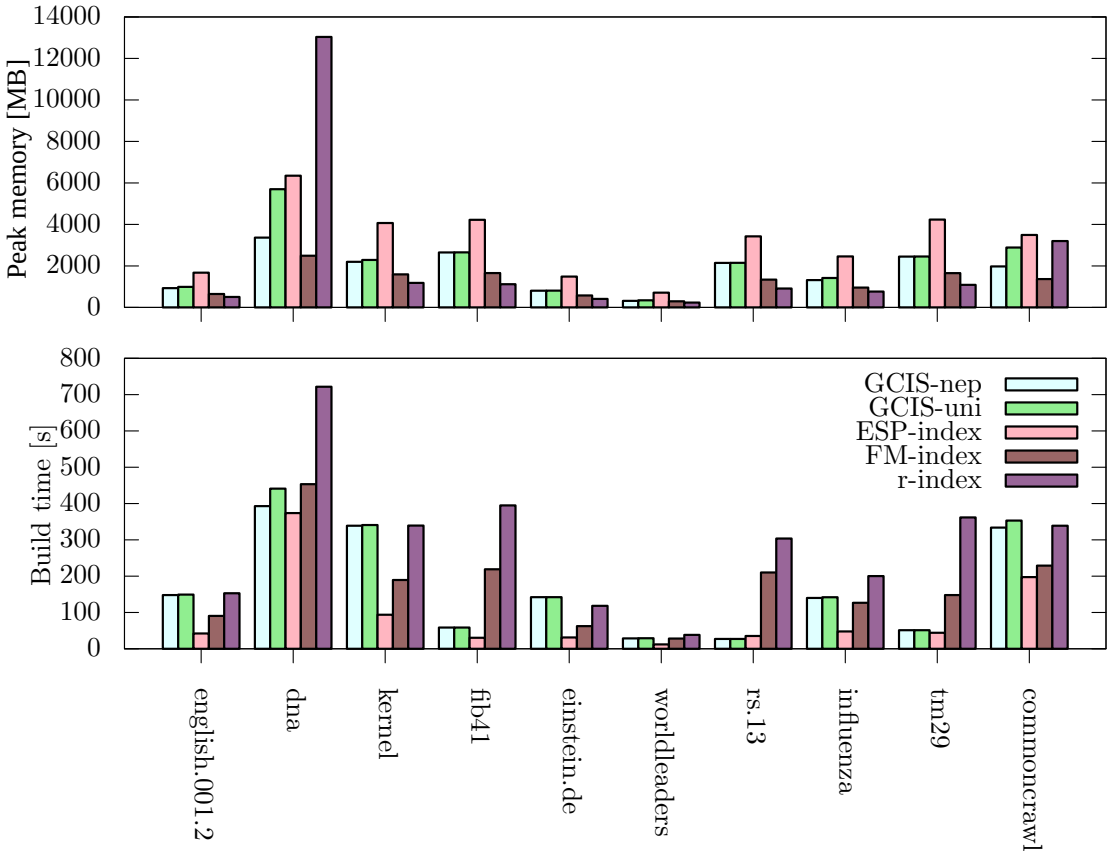

Figure 7: Maximum memory consumption *(top)* and time *(bottom)* during the construction of the indexes.

for decoding. In particular for ENGLISH.001.2, GCIS-nep is the fastest index when the pattern length reaches 10000 characters and more. At this pattern length, the pattern grammar reached a height τ_P of almost six, which is the height τ_T. The algorithm can extend an occurrence of a core to a pattern occurrence by checking only 80–100 characters. However, when the pattern surpasses 5000 characters, the computation of $\mathcal{G}_P$ becomes the time bottleneck. In that respect, the ESP-index shares the same characteristic. The universal encoding slows down locate time by roughly a factor of 2 to 10. Let us have a look at the dataset FIB41, which is linearly recurrent [15], a property from which we can derive the fact that a pattern that occurs at least once in T has actually a huge number of occurrences in T. There are almost 3,000,000 occurrences for patterns of length 100 in total. Here, we observe that our indexes are faster than ESP-index. ESP-index needs more time for locate than GCIS because GCIS can form a core that covers a higher percentage of the pattern than the core selected by ESP. FM-index, and ESP-index with $|P| = 10$ take 100 seconds or more on average – we omitted them in the graph to keep the visualization clear.

In Fig. 9, we study the average pattern grammar height $\tau_P = \mathcal{O}(\lg |P|)$. For this experiment, we created for each data point 100 patterns by randomly selecting a position j in T from which we extract a substring of the given length as a pattern. For every dataset, we could observe that τ_P is logarithmic in the pattern length, especially for the artificial datasets FIB41, TM29, and RS.13, where τ_P is empirically larger than measured in other datasets. In DNA and COMMONCRAWL, τ_P is at most 3, but this is because $\tau_T = 3$ for these datasets.

Next, we investigate the influence of the occurrences on the query time of locate on our indexes in Fig. 10. Instead of occ_C, which is almost always less than 5 for ENGLISH.001.2, we measure $\mathrm{occ}_C{}' \geq \mathrm{occ}_C$ with $\mathrm{occ}_C{}' = \mathcal{O}(\mathrm{occ}_C \lg n)$ being the number of all visited DAG nodes for finding the lowest ancestors of C whose expansion is large enough to extend that occurrence of C to an

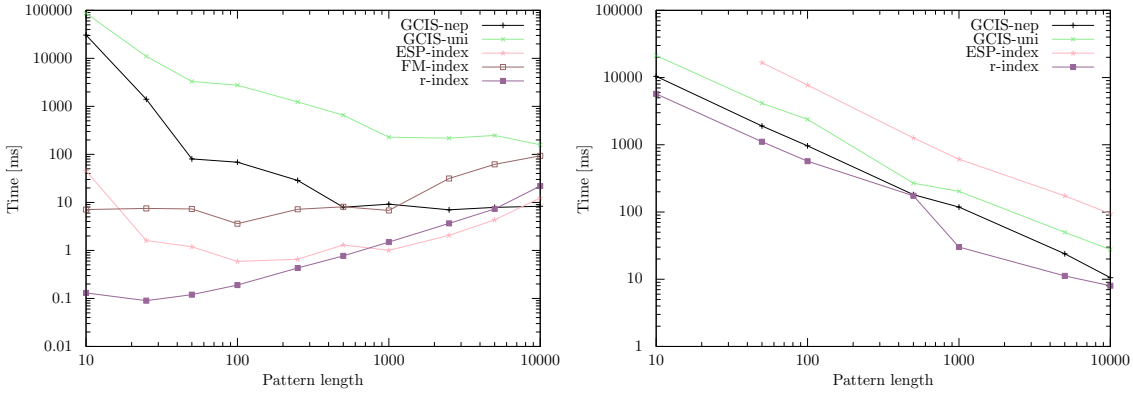

Figure 8: Time for locate while scaling the pattern length on the datasets ENGLISH.001.2 (left) and FIB41 (right). The plots are in logscale. The right figure does not feature the FM-index, which takes considerably more time than the other approaches. For the same reason, there is no data shown for the ESP-index for small pattern lengths, which needs 170 seconds on average for $|P| = 10$.

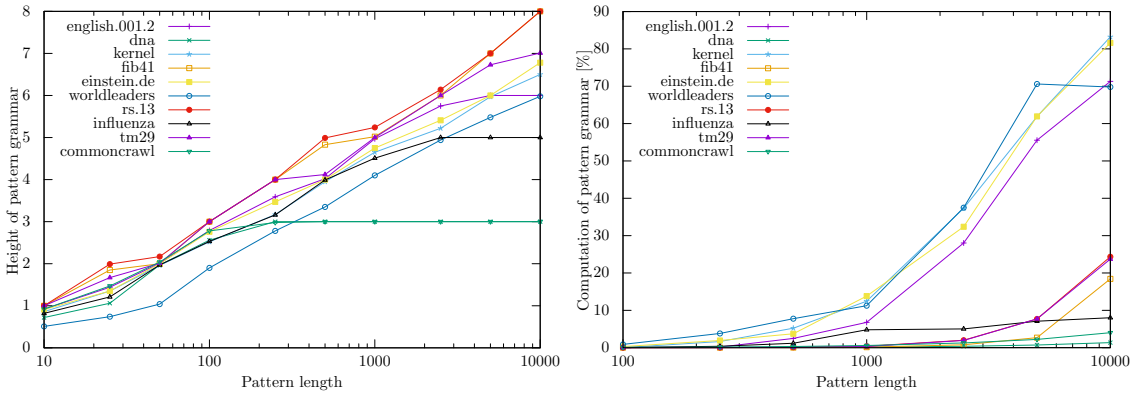

Figure 9: *Left:* The average height τ_P of $\mathcal{G}_P$ for a pattern of a certain length. *Right:* Percentage of the computation of $\mathcal{G}_P$ in relation to the whole running time for answering locate(P) with GCIS-nep.

occurrence of P. For that, we focus again on ENGLISH.001.2 for patterns of fixed length $|P| = 100$, where we observe that $\text{occ}_C{}'$ has a stronger influence on the running time than occ has.

As a practical optimization, we prematurely abort the computation of $\mathcal{G}_T$ at a certain height $\tau' < \tau_T$ when the reduction of $T^{(\tau'-1)}$ to $T^{(\tau')}$ with the newly introduced rules would increase the grammar size (we measure the needed space in terms of GCIS-nep). Hence, we may end up with a larger RHS of the start symbol $X^{(\tau')} \to T^{(\tau'-1)}$, which could be in fact the text itself if the text is incompressible with GCIS. This heuristic takes effect in non-highly-repetitive datasets such as DNA and COMMONCRAWL. It also has an effect on the height τ_P since the core must have a height less than the height of the start symbol of T. Without this heuristic, we reach a height $\tau_T = 9$ for DNA, and we also reach high values for τ_P, on average $\tau_P = 6.55$ for $|P| = 10000$, or $\tau_P = 5.81$ for $|P| = 5000$.

Finally, the parameter occ_C (core occurrences in grammar rules) is crucial for query performance. We expect small values of occ_C when the core corresponds to high-level non-terminals (large height h) and is used in few rules. As a rule of thumb we expect small occ_C values for large pattern lengths ($m \geq 1000$) on highly repetitive texts ($g < 0.1n$) when the pattern occurs in the text and the pattern grammar depth τ_P approaches the text grammar depth τ_T. In fact, under these conditions, GCIS index performs best, often outperforming alternatives by exploiting the small occ_C to avoid excessive verification work.

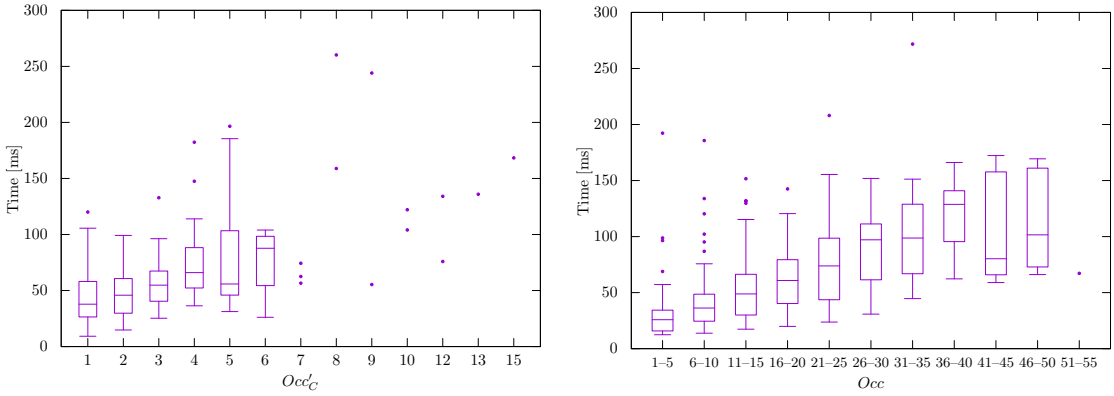

Figure 10: Relation of the number of visited DAG nodes $\mathrm{occ}_C{}'$ (left) or the number of occurrences of the pattern occ (right) with the time needed by GCIS-nep for answering locate(P) on the dataset ENGLISH.001.2 for patterns of fixed length $|P| = 100$.

9 Practical Observations and Analysis

We recall that our practical implementations GCIS-nep and GCIS-uni deviate from the theoretical algorithm in several ways:

- Premature stopping of grammar construction based on size heuristic, obtaining a height $\tau' < \tau_T$.
- Skipping the use of the generalized suffix tree GST altogether.
- Binary search on sorted RHSs for lookup.
- Linear scanning for finding core occurrences
- Naive character-by-character comparison instead of LCE: While the optimized LCE-based extension takes amortized $\mathcal{O}(m + \mathrm{occ}_C \tau_T)$ time, we use naive character-by-character comparison taking $\mathcal{O}(\mathrm{occ}_C m \tau_T)$ time.

9.1 Shorter Grammar Heights

For the first bullet point, we terminate grammar construction at height $\tau' < \tau_T$ if continuing would increase the grammar size as measured in the space metric of the GCIS-nep representation. As a rule of thumb, one could define a threshold $t \in (0, 1]$ and a weight $w_0 \geq 0$ (both real non-negative numbers) that give a criterion for stopping the grammar construction at height h when the compressed representation becomes larger than the previous level.

$$\frac{|T^{(h-1)}|}{|T^{(h)}| + w_0 |\Gamma^{(h)}|} < t.$$

The rationale is that the heuristic prevents pathological cases where recursive factorization creates many small non-terminals with high overhead. For incompressible or weakly compressible texts, forcing deeper factorization can expand g due to creating unique factors and thus non-repeated non-terminals. These non-terminals add additional overhead since we also need to keep track of their metadata (pointers, arrays) in the index. The practical impact of this heuristic differs based on dataset characteristics:

- DNA, COMMONCRAWL: Natural stopping at $\tau_T = 3$ due to low repetition
- ENGLISH.001.2: Stops at $\tau_T = 6$ balancing compression and overhead
- FIB41, TM29, RS.13: Deep recursion ($\tau_T = 20+$) provides exponential compression

For datasets where the heuristic stops early (small τ_T), patterns cannot achieve deep factorization, limiting the effectiveness of a core. However, as shown in Fig. 9, even with $\tau_T = 3$ on DNA, patterns reach $\tau_P = 2$–3 for $m \geq 1000$.

9.2 Performance Relative to Theory

On the one hand, the practical implementation tracks theoretical behavior in the following cases:

Long patterns with high compression For $m \geq 1000$ on highly repetitive texts (like FIB41, TM29), the pattern grammar depth τ_P approaches the text grammar depth τ_T. The core covers a large portion of the pattern, making occ_C small. As shown in Fig. 8, GCIS becomes faster than FM-index and r-index for $m \geq 5000$ on ENGLISH.001.2.

Moderate $\mathbf{occ}_C$ values When $\mathrm{occ}_C \leq 100$, the linear scan for core occurrences is a multiplicative factor in extension (practical $\mathcal{O}(m\tau_T)$ time compared to $\mathcal{O}(m + \tau_T)$ time in the theoretical analysis), which remains manageable.

Small grammar height For texts where $\tau_T \leq 10$, the depth-dependent factors ($\tau_T \lg g$ terms) have low constants. This holds for most real-world datasets in our experiments.

Cache-friendly access The sorted array representation of RHSs enables efficient linear scans with good cache locality, partially compensating the lack of the GST, which is used to find occurrences of C efficiently.

On the other hand, performance degrades relative to theory when:

Short patterns For $m < 100$, the $\mathcal{O}(g)$ scan time overhead dominates since g can be large even on compressed texts. We expect that an implementation achieving the theoretical bound of $\mathcal{O}(m \lg g)$ time would be faster.

Large $\mathbf{occ}_C$ When the core appears frequently in grammar rules ($\mathrm{occ}_C > 1000$), the $\mathcal{O}(\mathrm{occ}_C m\tau_T)$ extension time becomes significant. We expect that an implementation achieving the theoretical bound of $\mathcal{O}(m + \mathrm{occ}_C\tau_T)$ time would be faster.

$\mathbf{occ} \ll \mathbf{occ}_C$ The core-based approach is practical only when occ_C is not much larger than occ. One core occurrence in a rule can represent many text occurrences, so occ can be much larger than occ_C. Conversely, occ_C can be larger due to false candidates. Hence, there is no fixed ordering between occ and occ_C.

Empirical Validation Fig. 8 demonstrates that despite the simplifications, the implementation achieves the predicted theoretical advantage for long patterns on repetitive texts. The crossover point where GCIS becomes faster than FM-index/r-index occurs around $m = 5000$–10000, consistent with when the core-based approach provides meaningful savings.

10 Future Work

After determining the occ_C leaves in GST witnessing the occurrences of the core C of P in DAG, the rest of the algorithm is parallelizable since we process each of the GST leaves witnessing an occurrence of C individually, and may obtain $\mathcal{O}(m \lg |\mathcal{S}| + \mathrm{occ}_C \lg n \lg |\mathcal{S}|/p + \mathrm{occ}/p)$ time for running p processors in parallel. We could also partition the data structures on the grammar per height such that we manage each $\Gamma^{(h)}$ individually. This can practically improve the running time, and the space for GCIS-uni. Further, we would like to add a run-length compression step like in other run-length compressed grammars [27, 34]. Applying GCIS to the run-length encoding of the text would upper-bound the number of symbols of $F_1^{(h)}$ and $F_{z_h}^{(h)}$ to $2|\Gamma^{(h-1)}|$, or $2|\Sigma|$ for $h = 1$. We also would like to study the impact of DAC codes [2] for representing L, and compressed bit vectors such as [39] with rank/select support for the array Q.

Finally, the integer encoding with a hard-coded 32-bit representation limits the usage of our implementation to texts of medium size. For increasingly larger inputs, which are becoming more common nowadays, we want to enhance our implementation to support integers with larger bit-widths, which however requires a redesign of the underlying data structures. With a more flexible integer representation, we also want to enhance our experiments to compare with other grammar-compressed index implementations that can handle large inputs.

Acknowledgements

This article received funding from the Japan Society for the Promotion of Science (JSPS) KAKENHI with grant numbers JP23K24808, JP23K18466 (SI), JP25K00136 (YN), JP25K21150 (DK), and JP24K02899 (HB).

This work has been presented at LA Symposium Summer 2021, supported by the Research Institute for Mathematical Sciences, an International Joint Usage/Research Center located in Kyoto University.

References

[1] Tooru Akagi, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar index by induced suffix sorting. In *Proc. SPIRE*, volume 12944 of *LNCS*, pages 85–99, 2021.

[2] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Inf. Process. Manage.*, 49(1):392–404, 2013.

[3] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. STOC*, pages 279–288, 1998.

[4] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[5] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17 (1):8:1–8:39, 2021.

[6] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundam. Inform.*, 111(3):313–337, 2011.

[7] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 180–192, 2012.

[8] Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016.

[9] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *J. Comput. Syst. Sci.*, 118:53–74, 2021.

[10] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.

[11] Pierluigi Crescenzi, Alberto Del Lungo, Roberto Grossi, Elena Lodi, Linda Pagli, and Gianluca Rossi. Text sparsification via local maxima. *Theor. Comput. Sci.*, 304(1-3):341–364, 2003. doi: 10.1016/S0304-3975(03)00142-7.

[12] Jin Jie Deng, Wing-Kai Hon, Dominik Köppl, and Kunihiko Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. In *Proc. DCC*, pages 63–72, 2022. doi: 10.1109/DCC52660.2022.00014.

[13] Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. DCC*, pages 83–92, 2021.

[14] Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017.

[15] Chen Fei Du, Hamoon Mousavi, Luke Schaeffer, and Jeffrey O. Shallit. Decision algorithms for fibonacci-automatic words, with applications to pattern avoidance. *CoRR*, abs/1406.0670, 2014. URL `http://arxiv.org/abs/1406.0670`.

[16] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21 (2):246–260, 1974.

[17] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, 21(2):194–203, 1975.

[18] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[19] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[20] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1 – 1.12:31, 2008.

[21] Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting in the restore model. *ACM Trans. Algorithms*, 16(4):50:1–50:53, 2020.

[22] Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *Proc. LATA*, volume 7183 of *LNCS*, pages 240–251, 2012.

[23] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[24] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. *CoRR*, abs/1906.00809, 2019.

[25] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[26] Aaron Hong, Marco Oliva, Dominik Köppl, Hideo Bannai, Christina Boucher, and Travis Gagie. Acceleration of FM-index queries through prefix-free parsing. In *Proc. WABI*, volume 273 of *LIPIcs*, pages 13:1–13:16, 2023. doi: 10.4230/LIPIcs.WABI.2023.13.

[27] Artur Jez. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015.

[28] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. STOC*, pages 827–840, 2018.

[29] John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Information Theory*, 46(3):737–754, 2000.

[30] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[31] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305, 1999.

[32] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[33] Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.

[34] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[35] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discret. Appl. Math.*, 274:116–129, 2020.

[36] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[37] Daniel Saad Nogueira Nunes, Felipe Alves da Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. DCC*, pages 42–51, 2018.

[38] Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. Grammar compression by induced suffix sorting. *ACM J. Exp. Algorithmics*, 27:1.1:1–1.1:33, 2022. doi: 10.1145/3549992. URL `https://doi.org/10.1145/3549992`.

[39] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, 2007.

[40] Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *Proc. FOCS*, pages 320–328, 1996.

[41] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proc. SEA*, volume 8504 of *LNCS*, pages 338–350, 2014.

[42] Yoshimasa Takabatake, Kenta Nakashima, Tetsuji Kuboyama, Yasuo Tabei, and Hiroshi Sakamoto. siEDM: an efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26:1–26:18, 2016.

[43] Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar-compressed self-index with Lyndon words. *IPSJ TOM*, 13(2): 84–92, 2020.

[44] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.