Grammar Index by Induced Suffix Sorting

Tooru Akagi[1], Dominik Köppl[2(✉)], Yuto Nakashima[1], Shunsuke Inenaga[1,3], Hideo Bannai[2], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Fukuoka, Japan
{toru.akagi,yuto.nakashima,inenaga,takeda}@inf.kyushu-u.ac.jp

[2] M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan
{koeppl.dsc,hdbn.dsc}@tmd.ac.jp

[3] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Abstract. 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 side 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, where g is the sum of the lengths of the right hand sides of all production rules. We practically evaluate that our proposed index 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

Compressed text indexes have become the standard tool for maintaining highly-repetitive texts 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. One type of such self-indexes are *grammar indexes*, which are an augmentation of the admissible grammar [20] produced by a grammar compressor. Grammar indexes

© Springer Nature Switzerland AG 2021
T. Lecroq and H. Touzet (Eds.): SPIRE 2021, LNCS 12944, pp. 85–99, 2021.
https://doi.org/10.1007/978-3-030-86692-1_8

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.

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 sorting (GCIS) [29], which helps us to answer locate with an index built upon GCIS with the following bounds:

Theorem 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 $\mathcal{O}(m \lg |\mathcal{S}| + occ_C \lg n \lg |\mathcal{S}| + occ)$ 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 side 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 [25], ESP [7], HSP [16], the Rsync parse [17], or the grammar of [3, Sect. 4.2]. A brief review of these and other self-indexes follows.

1.2 Related Work

With respect to indexing a grammar for answering locate, the first work we are aware of is due to [5] who studied indices built upon so-called *straight-line programs (SLPs)*. An SLP is a context-free grammar representing a single string in the Chomsky normal form.

Other research focused on particular types of grammar, such as the ESP-index [24,31,32], an index [4] combining Re-Pair [22] with the Lempel–Ziv-77 parsing [34], a dynamic index [26] based on signature encoding [25], the Lyndon SLP [33], or the grammar index of [3]. For the experiments in Sect. 5, we will additionally have a look at other self-indexes capable of locate-queries. There, we analyze Burrows–Wheeler-transform (BWT) [2]-based approaches, namely the FM-index [15] and the r-index [18].

Finally, the grammar GCIS has other interesting properties besides being locality sensitive. [28] showed how to compute the suffix array and the longest-common-prefix array from GCIS during a decompression step restoring the original text. Recently, [8] showed how to compute the BWT directly from the GCIS grammar.

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 $\prec$.

2.1 Induced Suffix Sorting

SAIS [27] is a linear-time algorithm for computing the suffix array [23]. We briefly review the parts of SAIS important for constructing the GCIS grammar. SAIS assigns each suffix 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., $T[i..] \prec T[i+1..]$, where we stipulate that $T[|T|]$ 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 [27]) 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 $T[i+1]$.

The LMS substrings of #T for # being a special character smaller than all characters appearing in T such that #T starts with an S* position, induce a factorization of $T = F_1 \cdots F_z$, where each factor starts with an LMS substring. We call this factorization *LMS-factorization*. 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., $\pi^*(X) \subset \Sigma^*$ and $\pi^*(X^{(\tau_T)}) = T$. Since $\pi(X)$ is deterministically defined, we use to say *the right hand side of* X for $\pi(X)$.

Lemma 1 **([29]).** *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 $T^{(0)} = T$. Note that we use $\mathcal{T}_T$ only as a conceptional 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 $h > 1$ to a string of symbols on height $h - 1$. Hence, the grammar is acyclic.

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 [19, APL1]. The key difference is that we search the core of a pattern in the right hand sides of the rules. For that, we make use of the generalized suffix tree GST built upon the right hand sides 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. Each leaf λ stores a pointer to the non-terminal $X^{(h)}$ and an offset o such that $\pi(X^{(h)})[o..]$ 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 [1] in linear time and space in the number of nodes of GST. This data structure answers lca in constant time. Next, $\mathsf{child}(u, c)$ gives the child of the node u connected to u with an edge having a label starting with $c \in \Gamma$. 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, $\mathsf{string\_depth}(v)$ 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 $\mathsf{lookup}(S)$ 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 rule $X \in \Gamma$ 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 rule 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 right hand sides 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 $\mathsf{lce}(X, Y, i, j)$ 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 $\mathsf{lce}(X, Y, i, j)$ by selecting the leaves $X.P[i]$ and $Y.P[j]$, retrieve the LCA $\mathsf{lca}(X.P[i], Y.P[j])$ of both leaves, and take its string depth, all in constant time. More strictly speaking, we return $\min(|\pi(X)[i..]|, |\pi(Y)[j..]|, \mathsf{string\_depth}(\mathsf{lca}(X.P[i], Y.P[j])))$, since the delimiter \$ is not a unique character, but appears at each end of each right hand side in the underlying string R of GST.

Complexity Bounds. GST can be computed in $\mathcal{O}(g)$ time [13]. The grammar index consists of the GCIS grammar, GST built upon $|R| = g + |\Gamma|$ symbols, and augmented with a data structure for lca [1]. 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

Like [30, Sect. 2], our idea is to first fix a core C of a given pattern P, find the occurrences of C in the text, and then try to extend all these occurrences to occurrences of P.

4.1 Cores

A core C is a string of symbols of the GCIS grammar $\mathcal{G}_P$ built on the pattern P with the following property: given C consists of consecutive nodes on height $h \geq 0$ in $\mathcal{T}_T$, if there is an occurrence of C in $\mathcal{T}_T$ being a set of nodes on height h that have not the same parent node on height $h+1$, then the expansion of this occurrence of C does *not* lead to an occurrence of P. So for each occurrence of C in $\mathcal{T}_T$ whose expansion is contained in an occurrence of P, this occurrence is a (not necessarily proper) substring of the right hand side of a rule of $\mathcal{G}_T$.

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 an evidence of an occurrence of P.

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.

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, we want to assure that we only create a new non-terminal for a factor F whenever $\mathsf{lookup}(F) = \perp$; if $\mathsf{lookup}(F) = X$, we borrow the non-terminal X from $\mathcal{G}_T$. By doing so, we ensure that non-terminals of $\mathcal{G}_P$ and $\mathcal{G}_T$ are identical whenever their right hand sides of their productions are equal. In detail, if we create the factors $P^{(h)} = F_1^{(1)} \cdots F_{z_h}^{(h)}$, we first retrieve $Y_i^{(h)} := \mathsf{lookup}(P^{(h)})$ for each $i \in [2..z_h - 1]$. If one of the lookup-queries returns $\perp$, we abort since we can be sure that the pattern does not occur in T. That is because all non-terminals $Y_2^{(h)}, \ldots Y_{z_h-2}^{(h)}$ classify as cores. To see this, we observe that prepending or appending symbols to $P^{(h)}$ does not change the factors $F_2^{(h)}, \ldots, F_{z-1}^{(h)} =: C^{(h)}$.

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 $C^{(h)} = F_2^{(h)}, \ldots, F_{z-1}^{(h)}$. What we show is that we cannot change the type of any position $C^{(h)}[i]$ to $\mathtt{S}^*$: Firstly, the type of a position ($\mathtt{S}$ or $\mathtt{L}$) depends only on its succeeding position, and hence prepending cannot change the type of a position in $C^{(h)}$. 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 side, prepending character can only introduce a new factor or extend $F_1^{(h)}$. On the other side, 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.

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 $C_p C C_s$ with C_p and C_s possibly empty, and defined by one of the following mutually exclusive conditions: (1) If the LMS factorization consists of two non-empty factors $F_1 \cdot F_2$, then C_p is F_1. (2) Given $P^{(\tau_P)} = (P^{(\tau_P)}[j_1])^{c_1} \cdots (P^{(\tau_P)}[j_k])^{c_k}$ is the run-length-encoded representation of P with $1 = j_1 < \ldots < j_k = |P^{(\tau_P)}|$, $c_{j_i} \geq 1$ for $i \in [1..k]$, and $P[j_i] \neq P[j_{i+1}]$ for $i \in [1..k-1]$, we set $C_s \leftarrow (P^{(\tau_P)}[j_k])^{c_k}$ if $P^{(\tau_P)}[j_k] < P^{(\tau_P)}[j_{k-1}]$. (3) In the other cases, C_p and/or C_s are empty.

To see why C is a core, we only have 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. If C_s is empty, then C ends with P, and as a border case, the last position of C is $\mathtt{S}^*$. In that case, appending a symbol smaller than $P[m]$ to $F_1^{(h)}C$ changes the type of the last position of C to $\mathtt{L}$. If we append a symbol larger than $P[m]$, then the last position of C becomes $\mathtt{S}$, 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 total, there are symbols $A^{(1)}, \ldots, A^{(\tau_P-1)}$ and $S^{(1)}, \ldots, S^{(\tau_P-1)}$ such that

$$P = \pi(F_1^{(1)} \cdots F_1^{(\tau_P-2)} A^{(1)} \cdots A^{(\tau_P-2)} C S^{(\tau_P-2)} \cdots S^{(1)} F_{z_{\tau_P-2}}^{(\tau_P-2)} \cdots F_{z_1}^{(1)}), \quad (1)$$

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.

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 $\mathrm{occ}_C \in \mathcal{O}(g)$. This number is also the number of occurrences of C in the right hand sides 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. 1 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 right hand side 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 right hand sides of $\mathcal{G}_T$.

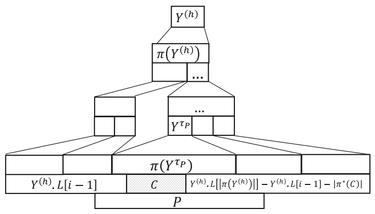

Fig. 1. Deriving a non-terminal $Y^{(h)}$ with $\pi^*(Y^{(h)})$ containing P from a non-terminal $Y^{(\tau_P)}$ with $\pi^*(Y^{(\tau_P)})$ containing $\pi^*(C)$. The expansion of none of the descendants of $Y^{(h)}$ towards $Y^{(\tau_P)}$ is large enough for extending its contained occurrence of $\pi^*(C)$ to an occurrence of P. We can check the expansion lengths of the substrings in $\pi(Y^{(h)})$ via the array $Y^{(h)}.L$.

We perform all operations in $\mathcal{O}(\mathrm{occ}_C \tau_T \lg |\mathcal{S}|)$ time because we query child in every recursion step.

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 $Y^{(\tau_P)}[o^{(\tau_P)}..]$ starts with C. By construction, these paths cover all occurrences of C in $\mathcal{T}_T$. Note that we process the DAG node $Y^{(\tau_P)}$ (but for different offsets $o^{(\tau_P)}$) as many times as C occurs in $\pi(Y^{(\tau_P)})$). In what follows, we try to expand the occurrence of C captured by $Y^{(\tau_P)}$ and $o^{(\tau_P)}$ to an occurrence of P.

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 maybe does not match P completely). For the explanation, we only focus on extending all occurrences of C to the right to CC_s (the left side side is done symmetrically). We maintain an array D of length τ_P storing pairs $(X^{(h)}, \ell_h)$ 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 one, 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 $\mathcal{O}(\mathrm{occ}_C \tau_T)$.

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 $X.L$ for that). 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 [6, 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 of climbing up the tree with the amount of occurrences occ of the pattern.

Total Time. To sum up, we spent $\mathcal{O}(m \lg |\mathcal{S}|)$ time for finding C, $\mathcal{O}(\mathrm{occ}_C \tau_T \lg |\mathcal{S}|)$ time for computing the non-terminals covering C, $\mathcal{O}(m + \mathrm{occ}_C \tau_T)$ time for reducing these non-terminals to W, and $\mathcal{O}(\mathrm{occ})$ 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.

5 Implementation and Experiments

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, 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 right hand sides, 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. 5.

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 right hand sides. To find the right hand side 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

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

Table 1. Sizes of the used datasets and the indexes stored on disk. Sizes are in megabytes [MB].

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	1.1458
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

order. Hence, we can evaluate $\mathsf{lookup}(S)$ 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 right hand sides contain C, we linearly scan the right hand sides of all non-terminals on height τ_P, which we can do cache-friendly since the right-hands 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 [21].

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 right hand sides 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 [11]. Next, we observe that each right hand side $\pi(X_i)$ form a bitonic sequence: the ranks of the first ℓ_i symbols are non-decreasing, while 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., $\varDelta[i][k] := |\pi(X_i)[k] - \pi(X_i)[k-1]|$ for $k \in [2..|\pi(X_i)|]$, which is stored in Elias-γ code [12]. 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 $\varDelta$, but need to adjust Q such that $Q[i]$ points to the first bit of $\varDelta[i]$. 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

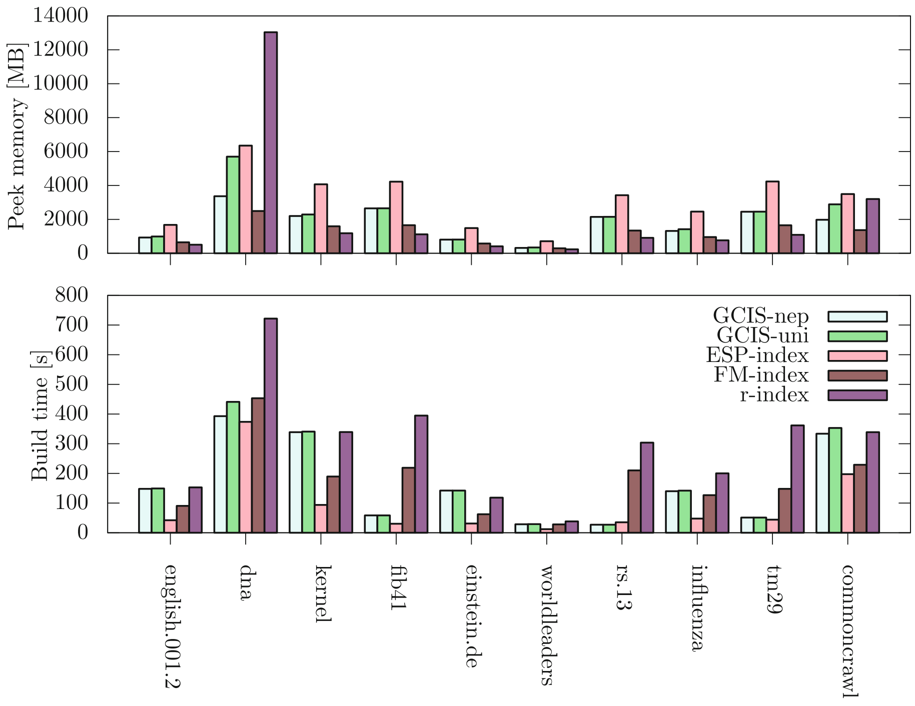

Fig. 2. Maximum memory consumption *(top)* and time *(bottom)* during the construction of the indexes.

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 linear 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.

Experiments. In the following we present an evaluation of our C++ implementation and different self-indexes for comparison, which are the FM-index [14], the ESP-Index [32], and the r-index [18][3]. All code has been compiled with `gcc-10.2.0` in the highest optimization mode `-O3`. We ran all our experiments with an Intel Xeon CPU X5670 clocked at 2.93 GHz running Arch Linux.

Our datasets shown in Table 1 are from the Pizza&Chili and the tudocomp [9] corpus.[4] With respect to the index sizes, we have the empirically ranking GCIS-uni < ESP-index < GCIS-nep, followed by one of the BWT-based indexes. While

[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.

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 2 gives the time and memory needed for constructing the indexes.

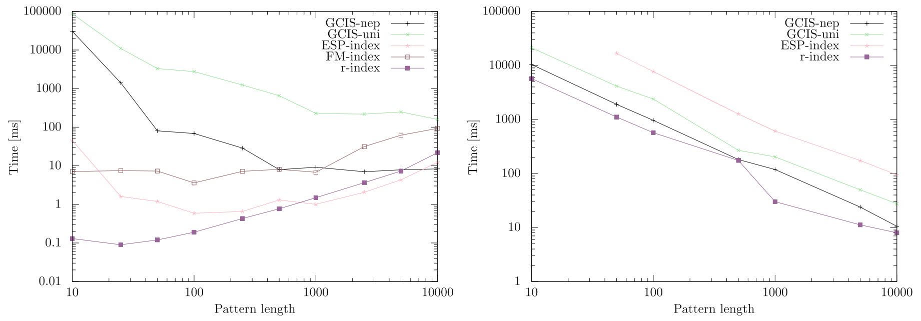

Fig. 3. Time for locate while scaling the pattern length on the datasets ENGLISH.001.2 (left) and FIB41 (right). The plots are in logscale. The right figure does not feature the FM-index, which takes considerably more time than the other approaches. For the same reason, there is no data shown for the ESP-index for small pattern lengths, which needs 170 s on average for $|P| = 10$.

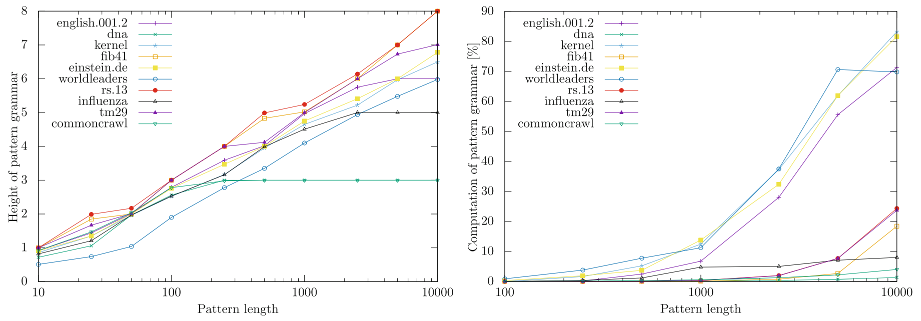

Fig. 4. *Left:* The average height τ_P of $\mathcal{G}_P$ for a pattern of a certain length. *Right:* Percentage of the computation of $\mathcal{G}_P$ in relation to the whole running time for answering locate(P) with GCIS-nep.

We can observe in Fig. 3 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 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 time, 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.

With that respect, the ESP-index shares the same characteristic. encoding make slow down the location time by about 2 to 10 times approximately. Let us have a look at the dataset FIB41, which is linearly recurrent [10], 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 occurrence of patterns with a length of 100. 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 than covers a higher percentage of the pattern than the core selected by ESP. FM-index, and ESP-index with $|P| = 10$ take 100 s or more on average – we omitted them in the graph to keep the visualization clear.

In Fig. 4, we study the maximum height $\tau_P = \mathcal{O}(\lg |P|)$ that we achieved for the patterns with $|P| = 100$ in each dataset. For this experiment, we randomly select a position j in T and extracted $P = T[j..j + 99]$. For every dataset, we could observe that τ_P is logarithmic to 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.

Acknowledgements. This work was supported by JSPS KAKENHI grant numbers JP21K17701 (DK), JP21K17705 (YN), JP20H04141 (HB), JP18H04098 (MT), and JST PRESTO grant number JPMJPR1922 (SI).

References

1. Buchsbaum, A.L., Kaplan, H., Rogers, A., Westbrook, J.R.: Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In: Proceedings of the STOC, pp. 279–288 (1998)
2. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, Palo Alto, California (1994)
3. Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. ACM Trans. Algorithms **17**(1), 8:1–8:39 (2021)
4. Claude, F., Fariña, A., Martínez-Prieto, M.A., Navarro, G.: Universal indexes for highly repetitive document collections. Inf. Syst. **61**, 1–23 (2016)
5. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundam. Inform. **111**(3), 313–337 (2011)
6. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 180–192. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34109-0_19
7. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. ACM Trans. Algorithms **3**(1), 2:1–2:19 (2007)
8. Díaz-Domínguez, D., Navarro, G.: A grammar compressor for collections of reads with applications to the construction of the BWT. In: Proceedings of the DCC, pp. 83–92 (2021)
9. Dinklage, P., Fischer, J., Köppl, D., Löbel, M., Sadakane, K.: Compression with the tudocomp framework. In: Proceedings of the SEA. LIPIcs, vol. 75, pp. 13:1–13:22 (2017)

10. Du, C.F., Mousavi, H., Schaeffer, L., Shallit, J.O.: Decision algorithms for fibonacci-automatic words, with applications to pattern avoidance. CoRR abs/1406.0670 (2014). http://arxiv.org/abs/1406.0670
11. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974)
12. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theory **21**(2), 194–203 (1975)
13. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000)
14. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: from theory to practice. ACM J. Exp. Algorithmics **13**, 1.12:1-1.123:1 (2008)
15. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the FOCS, pp. 390–398 (2000)
16. Fischer, J., I, T., Köppl, D.: Deterministic sparse suffix sorting in the restore model. ACM Trans. Algorithms **16**(4), 50:1-50:53 (2020)
17. Gagie, T., I, T., Manzini, G., Navarro, G., Sakamoto, H., Takabatake, Y.: Rpair: Rescaling RePair with Rsync. CoRR arXiv:abs/1906.00809 (2019)
18. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proceedings of the SODA, pp. 1459–1477 (2018)
19. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
20. 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)
21. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
22. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proceedings of the DCC, pp. 296–305 (1999)
23. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
24. Maruyama, S., Nakahara, M., Kishiue, N., Sakamoto, H.: ESP-index: a compressed index based on edit-sensitive parsing. J. Discret. Algorithms **18**, 100–112 (2013)
25. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)
26. Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Dynamic index and LZ factorization in compressed space. Discret. Appl. Math. **274**, 116–129 (2020)
27. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput. **60**(10), 1471–1484 (2011)
28. Nunes, D.S.N., Louza, F.A., Gog, S., Ayala-Rincn, M., Navarro, G.: Grammar compression by induced suffix sorting (2020)
29. Nunes, D.S.N., da Louza, F.A., Gog, S., Ayala-Rincón, M., Navarro, G.: A grammar compression algorithm based on induced suffix sorting. In: Proceedings of the DCC, pp. 42–51 (2018)
30. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In: Proceedings of the FOCS, pp. 320–328 (1996)
31. Takabatake, Y., Nakashima, K., Kuboyama, T., Tabei, Y., Sakamoto, H.: siEDM: an efficient string index and search algorithm for edit distance with moves. Algorithms **9**(2), 26:1–26:18 (2016)

32. Takabatake, Y., Tabei, Y., Sakamoto, H.: Improved ESP-index: a practical self-index for highly repetitive texts. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 338–350. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_29
33. Tsuruta, K., Köppl, D., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Grammar-compressed self-index with Lyndon words. IPSJ TOM **13**(2), 84–92 (2020)
34. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)