Universal Compressed Text Indexing [1]

Gonzalo Navarro [2]

Center for Biotechnology and Bioengineering (CeBiB), Department of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Nicola Prezza [3]

Department of Computer Science, University of Pisa, Italy.
nicola.prezza@di.unipi.it

Abstract

The rise of repetitive datasets has lately generated a lot of interest in compressed self-indexes based on dictionary compression, a rich and heterogeneous family of techniques that exploits text repetitions in different ways. For each such compression scheme, several different indexing solutions have been proposed in the last two decades. To date, the fastest indexes for repetitive texts are based on the run-length compressed Burrows–Wheeler transform (BWT) and on the Compact Directed Acyclic Word Graph (CDAWG). The most space-efficient indexes, on the other hand, are based on the Lempel–Ziv parsing and on grammar compression. Indexes for more universal schemes such as collage systems and macro schemes have not yet been proposed. Very recently, Kempa and Prezza [STOC 2018] showed that all dictionary compressors can be interpreted as approximation algorithms for the smallest *string attractor*, that is, a set of text positions capturing all distinct substrings. Starting from this observation, in this paper we develop the first *universal* compressed self-index, that is, the first indexing data structure based on string attractors, which can therefore be built on top of any dictionary-compressed text representation. Let γ be the size of a string attractor for a text of length n. From known reductions, γ can be chosen to be asymptotically equal to any repetitiveness measure: number of runs in the BWT, size of the CDAWG, number of Lempel–Ziv phrases, number of rules in a grammar or collage system, size of a macro scheme. Our index takes $\mathcal{O}(\gamma \lg(n/\gamma))$ words of space and supports locating the occ occurrences of any pattern of length m in $\mathcal{O}(m \lg n + occ \lg^{\epsilon} n)$ time, for any constant $\epsilon > 0$. This is, in particular, the first index for general macro schemes and collage systems. Our result shows that the relation between indexing and compression is much deeper than what was previously thought: the simple property standing at the core of all dictionary compressors is sufficient to support fast indexed queries.

Key words: Repetitive sequences; Compressed indexes; String attractors

1 Introduction

Efficiently indexing repetitive text collections is becoming of great importance due to the accelerating rate at which repetitive datasets are being produced in domains such as biology (where the number of sequenced individual genomes is increasing at an accelerating pace) and the web (with databases such as Wikipedia and GitHub being updated daily by thousands of users). A self-index on a string S is a data structure that offers direct access to any substring of S (and thus it replaces S), and at the same time supports indexed queries such as counting and locating pattern occurrences in S. Unfortunately, classic self-indexes — for example, the FM-index [20] — that work extremely well on standard datasets fail on repetitive collections in the sense that their compression rate does not reflect the input's information content. This phenomenon can be explained in theory with the fact that entropy compression is not able to take advantage of repetitions longer than (roughly) the logarithm of the input's length [22]. For this reason, research in the last two decades focused on self-indexing based on *dictionary compressors* such as the Lempel–Ziv 1977 factorization (LZ77) [39], the run-length encoded Burrows–Wheeler transform (RLBWT) [11] and context-free grammars (CFGs) [37], just to name the most popular ones. The idea underlying these compression techniques is to break the text into phrases coming from a dictionary (hence the name *dictionary compressors*), and to represent each phrase using limited information (typically, a pointer to other text locations or to an external set of strings). This scheme allows taking full advantage of long repetitions; as a result, dictionary-compressed self-indexes can be orders of magnitude more space-efficient than entropy-compressed ones on repetitive datasets.

The landscape of indexes for repetitive collections reflects that of dictionary compression strategies, with specific indexes developed for each compression strategy. Yet, a few main techniques stand at the core of most of the indexes. To date, the fastest indexes are based on the RLBWT and on the Compact Directed Acyclic Word Graph (CDAWG) [10,18]. These indexes achieve optimal-time queries (i.e., asymptotically equal to those of suffix trees [50]) at the price of a space consumption higher than that of other compressed indexes. Namely, the former index [26] requires $\mathcal{O}(r \lg(n/r))$ words of space, r being the number of equal-letter runs in the BWT of S, while the latter [3] uses $\mathcal{O}(e)$ words, e being the size of the CDAWG of S. These two measures (especially e) have

[1] This work started during Shonan Meeting 126 "Computation over Compressed Structured Data".

[2] Partially funded by Fondecyt Grant 1-170048 and Basal Funds FB0001, Chile.

[3] This work was partially done while the author was holding a post-doc position at the Technical University of Denmark (DTU). Partially funded by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

been experimentally confirmed to be not as small as others — such as the size of LZ77 — on repetitive collections [4].

Better measures of repetitiveness include the size z of the LZ77 factorization of S, the minimum size g of a CFG (i.e., sum of the lengths of the right-hand sides of the rules) generating S, or the minimum size g_{rl} of a run-length CFG [44] generating S. Indexes using $\mathcal{O}(z)$ or $\mathcal{O}(g)$ space do exist, but optimal-time queries have not yet been achieved within this space. Kreft and Navarro [38] introduced a self-index based on LZ77 compression, which proved to be extremely space-efficient on highly repetitive text collections [15]. Their self-index uses $\mathcal{O}(z)$ space and finds all the *occ* occurrences of a pattern of length m in time $\mathcal{O}((m^2 h + (m + occ) \lg z) \lg(n/z))$, where $h \leq z$ is the maximum number of times a symbol is successively copied along the LZ77 parsing. A string of length ℓ is extracted in $\mathcal{O}(h\ell)$ time. Similarly, self-indexes of size $\mathcal{O}(g)$ building on grammar compression [16,17] can locate all *occ* occurrences of a pattern in $\mathcal{O}(m^2 \lg\lg n + m \lg z + occ \lg z)$ time. Within this space, a string of length ℓ can be extracted in time $\mathcal{O}(\lg n + \ell / \lg_\sigma n)$ [7]. Alternative strategies based on Block Trees (BTs) [5] appeared recently. A BT on S uses $\mathcal{O}(z \lg(n/z))$ space, which is also the best asymptotic space obtained with grammar compressors [13,48,29,30,47]. In exchange for using more space than LZ77 compression, the BT offers fast extraction of substrings: $\mathcal{O}((1 + \ell / \lg_\sigma n) \lg(n/z))$ time. A self-index based on BTs has recently been described by Navarro [42]. Various indexes based on combinations of LZ77, CFGs, and RLBWTs have also been proposed [23,24,4,43,8,14]. Some of their best results are $\mathcal{O}(z \lg(n/z) + z \lg\lg z)$ space with $\mathcal{O}(m + occ(\lg\lg n + \lg^\epsilon z))$ query time [14], and $\mathcal{O}(z \lg(n/z))$ space with either $\mathcal{O}(m \lg m + occ \lg\lg n)$ [24] or $\mathcal{O}(m + \lg^\epsilon z + occ(\lg\lg n + \lg^\epsilon z))$ [14] query time. Gagie et al. [26] give a more detailed survey.

The above-discussed compression schemes are the most popular, but not the most space-efficient ones. More powerful compressors (NP-complete to optimize) include macro schemes [49] and collage systems [36]. Not much work exists in this direction, and no indexes are known for these particular compressors.

1.1 String attractors

As seen in the previous paragraphs, the landscape of self-indexes based on dictionary compression — as well as that of dictionary compressors themselves — is extremely fragmented, with several techniques being developed for each distinct compression strategy. Very recently, Kempa and Prezza [35] gathered all dictionary compression techniques under a common theory: they showed that these algorithms are approximations to the smallest *string attractor*, that is, a set of text positions “capturing” all distinct substrings of S.

Definition 1 (String attractor [35]). A *string attractor* of a string $S[1..n]$ is a set of γ positions $\Gamma = \{j_1, \ldots, j_\gamma\}$ such that every substring $S[i..j]$ has an occurrence $S[i'..j'] = S[i..j]$ with $j_k \in [i', j']$, for some $j_k \in \Gamma$.

Their main result is a set of reductions from dictionary compressors to string attractors of asymptotically the same size.

Theorem 1 ([35]). *Let S be a string and let α be any of these measures:*

(1) the size g of a CFG for S,
(2) the size g_{rl} of a run-length CFG for S,
(3) the size c of a collage system for S,
(4) the size z of the LZ77 parse of S,
(5) the size b of a macro scheme for S.

Then, S has a string attractor of size $\gamma = \mathcal{O}(\alpha)$. In all cases, the corresponding attractor can be computed in $\mathcal{O}(|S|)$ time and space from the compressed representation.

Importantly, this implies that any data structure based on string attractors is *universal*: given any dictionary-compressed text representation, we can induce a string attractor and build the data structure on top of it. Indeed, the authors exploit this observation and provide the first universal data structure for random access, of size $\mathcal{O}(\gamma \lg(n/\gamma))$. Their extraction time within this space is $\mathcal{O}(\lg(n/\gamma) + \ell/\lg_\sigma n)$. By using slightly more space, $\mathcal{O}(\gamma \lg(n/\gamma) \lg^\epsilon n)$ for any constant $\epsilon > 0$, they obtain time $\mathcal{O}(\lg(n/\gamma)/\lg\lg n + \ell/\lg_\sigma n)$, which is the optimal that can be reached using any space in $\mathcal{O}(\gamma \operatorname{polylog} n)$ [35]. This suggests that compressed computation can be performed independently from the compression method used while at the same time matching the lower bounds of individual compressors (at least for some queries such as random access).

1.2 Our Contributions

In this paper we exploit the above observation and describe *the first universal self-index* based on string attractors, that is, the first indexing strategy not depending on the underlying compression scheme. Since string attractors stand at the core of the notion of compression, our result shows that the relation between compression and indexing is much deeper than what was previously thought: the simple string attractor property introduced in Definition 1 is sufficient to support indexed pattern searches.

Theorem 2. *Let a string $S[1..n]$ have an attractor of size γ. Then, for any constant $\epsilon > 0$, there exists a data structure of size $\mathcal{O}(\gamma \lg(n/\gamma))$ that, given a pattern string $P[1..m]$, outputs all the occ occurrences of P in S in time*

$\mathcal{O}(m \lg n + occ(\lg^\epsilon \gamma + \lg\lg(n/\gamma))) = \mathcal{O}(m \lg n + occ \lg^\epsilon n)$. *It can be built in* $\mathcal{O}(n + \gamma \lg(n/\gamma)\sqrt{\lg \gamma})$ *worst-case time and* $\mathcal{O}(\gamma \lg(n/\gamma))$ *space with a Monte Carlo method returning a correct index with high probability. A guaranteed construction, using Las Vegas randomization, takes* $\mathcal{O}(n \lg n)$ *expected time (this time also holds w.h.p.) and* $\mathcal{O}(n)$ *space.*

We remark that no representation offering random access within $o(\gamma \lg(n/\gamma))$ space is known. The performance of our index is close to that of the fastest self-indexes built on other repetitiveness measures, and it is the first one that can be built for macro schemes and collage systems.

To obtain our results, we adapt the block tree index of Navarro [42], which is designed for block trees on the LZ77 parse, to operate on string attractors. The result is also different from the block-tree-like structure Kempa and Prezza use for extraction [35], because that one is aligned with the attractors and this turns out to be unsuitable for indexing. Instead, we use a block-tree-like structure, which we dub Γ-tree, which partitions the text in a regular form. We moreover introduce recent techniques [24] to remove the quadratic dependency on the pattern length in query times.

1.3 Notation

We denote by $S[1..n] = S[1] \cdots S[n]$ a string of length n over an alphabet of size $\sigma = \mathcal{O}(n)$. Substrings of S are denoted $S[i..j] = S[i] \cdots S[j]$, and they are called prefixes of S if $i = 1$ and suffixes of S if $j = n$. The concatenation of strings S and S' is denoted $S \cdot S'$. We assume the RAM model of computation with a computer word of $\omega = \Omega(\lg n)$ bits. By lg we denote the logarithm function, to the base 2 when this matters.

The term *with high probability* (*w.h.p.* abbreviated) indicates with probability at least $1 - n^{-c}$ for an arbitrarily large constant c, where n is the input size (in our case, the input string length).

In our results, we make use of a modern variant of Karp–Rabin fingerprinting [33] (more common nowadays than the original version), defined as follows. Let $q \geq \sigma$ be a prime number, and r be a uniform number in $[1..q-1]$. The fingerprint $\hat{\phi}$ of a string $S = S[1] \cdots S[n] \in [1..\sigma]^n$ is defined as $\hat{\phi}(S) = \sum_{i=0}^{n-1} S[n-i] \cdot r^i \mod q$. The *extended fingerprint* of S is the triple $\phi(S) = \langle \hat{\phi}(S), r^{|S|} \mod q, r^{-|S|} \mod q \rangle$. We say that $S \neq S'$, with $S, S' \in [1..\sigma]^n$, collide through ϕ (for our purposes, it will be sufficient to consider equal-length strings) if $\hat{\phi}(S) = \hat{\phi}(S')$, that is, $\hat{\phi}(S - S') = 0$, where $S'' = S - S'$ is the string defined as $S''[i] = S[i] - S'[i] \mod q$. Since $\hat{\phi}(S'')$ is a polynomial (in the variable r) of degree at most $n-1$ in the field $\mathcal{Z}_q$, it has at most $n-1$ roots. As a consequence, the probability of having a collision between two strings is bounded by $\mathcal{O}(n/q)$ when r is uniformly chosen in $[1..q-1]$. By choosing $q \in \Theta(n^{c+2})$ for an arbitrarily large constant c, one obtains that such a hash function is collision-free among all equal-length substrings of a given string S of length n w.h.p. To conclude, we will exploit the (easily provable) folklore fact that two extended fingerprints $\phi(U)$ and $\phi(V)$ can be combined in constant time to obtain the extended fingerprint $\phi(UV)$. Similarly, $\phi(UV)$ and $\phi(U)$ (respectively, $\phi(V)$) can be combined in constant time to obtain $\phi(V)$ (respectively, $\phi(U)$). From now on, we will by default use the term "fingerprint" to mean extended fingerprint.

2 Γ-Trees

Given a string $S[1..n]$ over an alphabet $[1..\sigma]$ with an attractor Γ of size γ, we define a Γ-tree on S as follows. At the top level, numbered $l = 0$, we split S into γ substrings (which we call blocks) of length $b_0 = n/\gamma$. Each block is then recursively split into two so that if b_l is the length of the blocks at level l, then it holds that $b_{l+1} = b_l/2$, until reaching blocks of one symbol after $\lg(n/\gamma)$ levels (that is, $b_l = n/(\gamma \cdot 2^l)$).[4] At each level l, every block that is at distance $< b_l$ from a position $j \in \Gamma$ is *marked* (the distance between j and a block $S[i..i']$ is $i - j$ if $i > j$, $j - i'$ if $i' < j$, and 0 otherwise). Blocks $S[i..i']$ that are not marked are replaced by a pointer $\langle ptr_1, ptr_2, \delta \rangle$ to an occurrence $S[j'..j'']$ of $S[i..i']$ that includes a position $j \in \Gamma$, $j' \le j \le j''$. Such an occurrence exists by Definition 1. Moreover, it must be covered by 1 or 2 consecutive marked blocks of the same level due to our marking mechanism, because all the positions in $S[j'..j'']$ are at distance $< b_l$ from j. Those 1 or 2 nodes of the Γ-tree are ptr_1 and ptr_2, and δ is the offset of j' within ptr_1 ($\delta = 0$ if j' is the first symbol inside ptr_1).

In level $l + 1$ we explicitly store only the children of the blocks that were marked in level l. The blocks stored in the Γ-tree (i.e., all blocks at level 0 and those having a marked parent) are called *explicit*. In the last level, the marked blocks store their corresponding single symbol from S.

See Figure 1 for an example of a Γ-tree. We can regard the Γ-tree as a binary tree (with the first $\lg \gamma$ levels chopped out), where the internal nodes are marked nodes and have two children, whereas the leaves either are marked and represent just one symbol, or are unmarked and represent a pointer to 1 or 2 marked nodes of the same level. If we call w the number of leaves, then there are $w - \gamma$ (marked) internal nodes. From the leaves, γ of them represent single symbols in the last level, while the other $w - \gamma$ leaves are unmarked blocks replaced by pointers. Thus, there are in total $2w - \gamma$ nodes in the tree,

[4] For simplicity of description, we assume that n/γ is a power of 2.

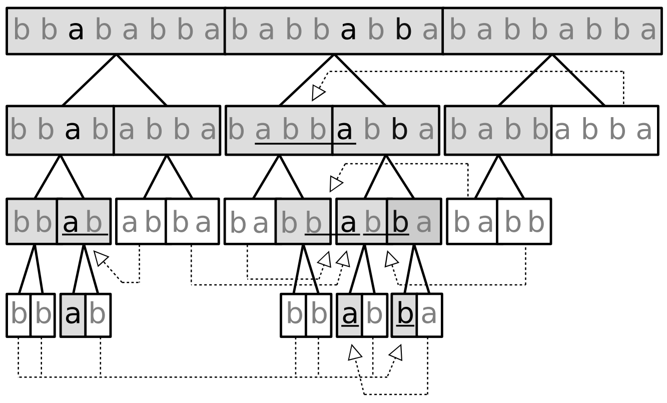

Fig. 1. Example of a Γ-tree built on a text of length $n = 24$ with $\gamma = 3$ attractor positions (black letters). Marked blocks are colored in gray. Each non-marked block (in white) is associated with an occurrence (underlined) crossing an attractor position, and therefore overlapping only marked blocks. Only explicit blocks are shown.

of which $w - \gamma$ are internal nodes, $w - \gamma$ are pointers, and γ store explicit symbols. Alternatively, w nodes are marked (internal nodes plus leaves) and $w - \gamma$ are unmarked (leaves only).

The Γ-tree then uses $\mathcal{O}(w)$ space. To obtain a bound in terms of n and γ, note that, at each level, each $j \in \Gamma$ may mark up to 3 blocks; thus there are $w \leq 3\gamma \lg(n/\gamma)$ marked blocks in total and the Γ-tree uses $\mathcal{O}(\gamma \lg(n/\gamma))$ space.

We now describe two operations on Γ-trees that are fundamental to support efficient indexed searches. The former is also necessary for a self-index, as it allows us extracting arbitrary substrings of S efficiently. We remind that this procedure is not the same described on the original structure of string attractors [35], because the structures are also different.

2.1 Extraction

To extract a single symbol $S[i]$, we first map it to a local offset $1 \leq i' \leq b_0$ in its corresponding level-0 block. In general, given the local offset i' of a character in the current block at level l, we first see if the current block is marked. If so, we map i' to a position in the next level $l + 1$, where the current block is split into two blocks of half the length: if $i' \leq b_{l+1}$, then we continue on the left child with the same offset; otherwise, we subtract b_{l+1} from i' and continue on the right child. If, instead, i is not in a marked block, we take the pointer $\langle ptr_1, ptr_2, \delta \rangle$ stored for that block and add δ to i'. If the result is $i' \leq b_l$, then we continue in the node ptr_1 with offset i'; otherwise, we continue in ptr_2 with the offset $i' - b_l$. In both cases, the new node is marked, so we proceed as on marked blocks in order to move to the next level in constant time. The total time to extract a symbol is then $\mathcal{O}(\lg(n/\gamma))$.

A substring of length ℓ can thus be extracted in time $\mathcal{O}(\ell \lg(n/\gamma))$, which will be sufficient to obtain the search time complexity of Theorem 2. It is possible to augment Γ-trees to match the complexity $\mathcal{O}(\lg(n/\gamma) + \ell / \lg_\sigma n)$ obtained by Kempa and Prezza [35] on string attractors as well, though this would have no impact on our results.

2.2 Fingerprinting

We now show that the Γ-tree can be augmented to compute the Karp–Rabin fingerprint of any text substring in logarithmic time.

Lemma 1. *Let $S[1..n]$ have an attractor of size γ and ϕ a Karp–Rabin fingerprint function. Then we can store a data structure of size $\mathcal{O}(\gamma \lg(n/\gamma))$ words supporting the computation of ϕ on any substring of S in $\mathcal{O}(\lg(n/\gamma))$ time.*

Proof. We augment our Γ-tree of S as follows. At level 0, we store the Karp–Rabin fingerprints of all the text prefixes ending at positions $i \cdot n/\gamma$, for $i = 1, \ldots, \gamma$. At levels $l > 0$, we store the fingerprints of all explicit blocks.

We first show that we can reduce the problem to that of computing the fingerprints of two prefixes of explicit blocks. Then, we show how to solve the sub-problem of computing fingerprints of prefixes of explicit blocks.

Let $S[i..j]$ be the substring of which we wish to compute the fingerprint $\phi(S[i..j])$. Note that $\phi(S[i..j])$ can be computed in constant time from $\phi(S[1..i-1])$ and $\phi(S[1..j])$ so we can assume, without loss of generality, that $i = 1$ (i.e., the substring is a prefix of S). Then, at level 0 the substring spans a sequence $B_1 \cdots B_t$ of blocks followed by a prefix C of block B_{t+1} (the sequence of blocks or C could be empty). The fingerprint of $B_1 \cdots B_t$ is explicitly stored, so the problem reduces to that of computing the fingerprint of C.

We now show how to compute the fingerprint of a prefix of an explicit block (at any level) in $\mathcal{O}(\lg(n/\gamma))$ time. We distinguish two cases.

(A) We wish to compute the fingerprint of $B[1..k]$, for some $k \le b_l$, and B is a marked block at level l. Let B_{left} and B_{right} be the children of B at level $l+1$. Then, the problem reduces to either (i) computing the fingerprint of $B_{\mathit{left}}[1..k]$ if $k \le b_{l+1}$, or combining the fingerprints of B_{left} (which is stored) and $B_{right}[1..k - b_{l+1}]$. In both sub-cases, the problem reduces to that of computing the fingerprint of the prefix of a block at level $l + 1$, which is explicit since B is marked.

(B) We wish to compute the fingerprint of $B[1..k]$, for some $k \leq b_l$, but B is an unmarked explicit block. Then, B is linked (through a Γ-tree pointer) to an occurrence in the same level spanning at most two blocks, both of which are marked. If the occurrence of B spans only one marked block B' at level l, then $B[1..b_l] = B'[1..b_l]$ and we are back in case (A). Otherwise, the occurrence of B spans two marked blocks B' and B'' at level l: $B[1..b_l] = B'[i..b_l]B''[1..i - 1]$, with $i \leq b_l$. For each pointer of this kind in the Γ-tree, we store the fingerprint of $B'[i..b_l]$. We consider two sub-cases. (B.1) If $k \geq b_l - i + 1$, then $B[1..k] = B'[i..b_l]B''[1..k - (b_l - i + 1)]$. Since we store the fingerprint of $B'[i..b_l]$, the problem reduces again to that of computing the fingerprint of the prefix $B''[1..k - (b_l - i + 1)]$ of a marked (explicit) block. (B.2) If $k < b_l - i + 1$, then $B[1..k] = B'[i..i + k - 1]$. Although this is not a prefix nor a suffix of a block, note that $B[1..k]B'[i + k..b_l] = B'[i..i + k - 1]B'[i + k..b_l] = B'[i..b_l]$. It follows that we can retrieve the fingerprint of $B[1..k]$ in constant time using the fingerprints of $B'[i+k..b_l]$ and $B'[i..b_l]$. The latter value is explicitly stored. The former is the fingerprint of the suffix of an explicit (marked) block. In this case, note that the fingerprint of a block's suffix can be retrieved from the fingerprint of the block and the fingerprint of a block's prefix, so we are back to the problem of computing the fingerprint of an explicit block's prefix.

To sum up, computing a prefix of an explicit block at level l reduces to the problem of computing a prefix of an explicit block at level $l + 1$ (plus a constant number of arithmetic operations to combine values). In the worst case, we navigate down to the leaves, where fingerprints of single characters can be computed in constant time. Combining this procedure into our main algorithm, we obtain the claimed running time of $\mathcal{O}(\lg(n/\gamma))$. □

3 A Universal Self-Index

Our self-index structure builds on the Γ-tree of S. It is formed by two main components: the first finds all the pattern occurrences that cross explicit block boundaries, whereas the second finds the occurrences that are completely inside unmarked blocks.

Lemma 2. *Any substring $S[i..j]$ of length at least 2 either overlaps two consecutive explicit blocks or is completely inside an unmarked block.*

Proof. The leaves of the Γ-tree partition S into a sequence of explicit blocks: γ of those are attractor positions and the other $w - \gamma$ are unmarked blocks.

Clearly, if $S[i..j]$ is not completely inside an unmarked block, it must cross a boundary between two explicit blocks. $\square$

We exploit the lemma in the following way. We will define an occurrence of P as *primary* if it overlaps two consecutive explicit blocks. The occurrences that are completely contained in an unmarked block are *secondary*. By the lemma, every occurrence of P is either primary or secondary. We will use a data structure to find the primary occurrences and another one to detect the secondary ones. The primary occurrences are found by exploiting the fact that a prefix of P matches at the end of an explicit block and the remaining suffix of P matches the text that follows. Secondary occurrences, instead, are found by detecting primary or other secondary occurrences within the area where an unmarked block points.

We note that this idea is a variant of the classical one [32] used in all indexes based on LZ77 and CFGs. Now we show that the principle can indeed be applied on attractors, which is the general concept underlying all those compression methods (and others where no indexes exist yet), and therefore unifies all those particular techniques.

3.1 Primary Occurrences

We describe the data structures and algorithms used to find the primary occurrences. Overall, they require $\mathcal{O}(w)$ space and find the occ_p primary occurrences in time $\mathcal{O}(m \lg(mn/\gamma) + occ_p \lg^\epsilon w)$, for any constant $\epsilon > 0$.

Data Structures. The leaves of the Γ-tree partition S into explicit blocks. The partition is given by the starting positions $1 = p_1 < \ldots < p_w \leq n$ of the leaves. By Lemma 2, every primary occurrence contains some substring $S[p_i - 1..p_i]$ for $1 < i \leq w$.

If we find the occurrences considering only their leftmost covered position of the form $p_i - 1$, we also ensure that each primary occurrence is found once. Thus, we will find primary occurrences as a prefix of P appearing at the end of some $S[p_{i-1}..p_i - 1]$ followed by the corresponding suffix of P appearing at the beginning of $S[p_i..n]$. For this sake, we define the set of pairs $\mathcal{B} = \{\langle S[p_i..n], S[p_{i-1}..p_i - 1]^{rev}\rangle,\ 1 < i \leq w\}$, where $S[p_{i-1}..p_i - 1]^{rev}$ means $S[p_{i-1}..p_i - 1]$ read backwards, and form multisets $\mathcal{X}$ and $\mathcal{Y}$ with the left and right components of $\mathcal{B}$, respectively.

We then lexicographically sort $\mathcal{X}$ and $\mathcal{Y}$, to obtain the strings $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$. All the occurrences ending with a certain prefix of P will form a contiguous range in the sorted multiset $\mathcal{Y}$, whereas all those starting with a certain suffix of P will form a contiguous range in the sorted multiset $\mathcal{X}$. Each primary occurrence of P will then correspond to a pair $\langle X_x, Y_y \rangle \in \mathcal{B}$ where both X_x and Y_y belong to their range.

Our structure to find the primary occurrences is a two-dimensional discrete grid G storing one point (x, y) for each pair $\langle X_x, Y_y \rangle \in \mathcal{B}$. The grid G is of size $(w-1) \times (w-1)$. We represent G using a two-dimensional range search data structure requiring $\mathcal{O}(w)$ space [12] that reports the t points lying inside any rectangle of the grid in time $\mathcal{O}((t+1) \lg^{\epsilon} w)$, for any constant $\epsilon > 0$. We also store an array $T[1..w-1]$ that, for each point (x, y) in G, where $X_x = S[p_i..n]$, stores $T[y] = p_i$, that is, where X_x starts in S.

Queries. To search for a pattern $P[1..m]$, we first find its primary occurrences using G as follows. For each partition $P_< = P[1..k]$ and $P_> = P[k+1..m]$, for $1 \le k < m$, we search $\mathcal{Y}$ for $P_<^{rev}$ and $\mathcal{X}$ for $P_>$. For each identified range $[x_1, x_2] \times [y_1, y_2]$, we extract all the t corresponding primary occurrences (x, y) in time $\mathcal{O}((t+1) \lg^{\epsilon} w)$ with our range search data structure. Then we report a primary occurrence starting at $T[y] - k$ for each such point (x, y). Over the m intervals, this adds up to $\mathcal{O}((m + occ_p) \lg^{\epsilon} w) = \mathcal{O}((m + occ_p) \lg^{\epsilon}(\gamma \lg(n/\gamma)))$.

We obtain the $m-1$ ranges in the multiset $\mathcal{X}$ in overall time $\mathcal{O}(m \lg(mn/\gamma))$, by using the fingerprint-based technique of Gagie et al. [24] applied to the z-fast trie of Belazzougui et al. [2] (we use a lemma from Gagie et al. [26] where the overall result is stated in cleaner form). We use an analogous structure to obtain the ranges in $\mathcal{Y}$ of the suffixes of the reversed pattern.

Lemma 3 (adapted from [26, Lem. 5.2])**.** *Let $S[1..n]$ be a string on alphabet $[1..\sigma]$, $\mathcal{X}$ be a sorted set of suffixes of S, and ϕ a Karp–Rabin fingerprint function. If one can extract a substring of length ℓ from S in time $f_e(\ell)$ and compute ϕ on it in time $f_h(\ell)$, then one can build a data structure of size $\mathcal{O}(|\mathcal{X}|)$ that obtains the lexicographic ranges in $\mathcal{X}$ of the $m-1$ suffixes of a given pattern P in worst-case time $\mathcal{O}(m \lg(\sigma)/\omega + m(f_h(m) + \lg m) + f_e(m))$ — provided that ϕ is collision-free among substrings of S whose lengths are powers of two.*

In Sections 2.1 and 2.2 we show how to extract in time $f_e(\ell) = \mathcal{O}(\ell \lg(n/\gamma))$ and how to compute a fingerprint in time $f_h(\ell) = \mathcal{O}(\lg(n/\gamma))$, respectively. In Section 4.2 we show that a Karp–Rabin function that is collision-free among substrings whose lengths are powers of two can be efficiently found. Together, these results show that we can find all the ranges in $\mathcal{X}$ and $\mathcal{Y}$ in time $\mathcal{O}(m \lg(\sigma)/\omega + m(\lg(n/\gamma) + \lg m) + m \lg(n/\gamma)) = \mathcal{O}(m \lg(mn/\gamma))$.

Patterns P of length $m = 1$ can be handled as $P[1]*$, where $*$ stands for any character. Thus, we take $[x_1, x_2] = [1, w]$ and carry out the search as a normal pattern of length $m = 2$. To make this work also for the last position in S, we assume as usual that S is terminated by a special symbol \$ that cannot appear in search patterns P. Alternatively, we can store a list of the marked leaves where each alphabet symbol appears, and take those as the primary occurrences. A simple variant of Lemma 2 shows that occurrences of length 1 are either attractor positions or are inside an unmarked block, so the normal mechanism to find secondary occurrences from this set works correctly. Since there are in total γ such leaves, the space for these lists is $\mathcal{O}(\sigma + \gamma) = \mathcal{O}(\gamma)$.

3.2 Secondary Occurrences

We now describe the data structures and algorithms to find the secondary occurrences. They require $\mathcal{O}(w)$ space and find the occ_s secondary occurrences in time $\mathcal{O}((occ_p + occ_s) \lg \lg_\omega (n/\gamma))$.

Data structures. To track the secondary occurrences, let us call *target* and *source* the text areas $S[i..i']$ and $S[j'..j'']$, respectively, of an unmarked block and its pointer, so that there is some $j \in \Gamma$ contained in $S[j'..j'']$ (if the blocks are at level l, then $i' = i + b_l - 1$ and $j'' = j' + b_l - 1$). Let $S[pos..pos+m-1]$ be an occurrence we have already found (using the grid G, initially). Our aim is to find all the sources that contain $S[pos..pos+m-1]$, since their corresponding targets then contain other occurrences of P.

To this aim, we store the sources of all levels in an array $R[1..w - \gamma]$, with fields j' and j'', ordered by starting positions $R[k].j'$. We build a predecessor search structure on the fields $R[k].j'$, and a range maximum query (RMQ) data structure on the fields $R[k].j''$, able to find the maximum endpoint j'' in any given range of R. While a predecessor search using $\mathcal{O}(w)$ space requires $\mathcal{O}(\lg \lg_\omega (n/w))$ time on an ω-bit-word machine [45, Sec. 1.3.2], the RMQ structure operates in constant time using just $\mathcal{O}(w)$ bits [21].

Queries. Let $S[pos..pos+m-1]$ be a primary occurrence found. A predecessor search for pos gives us the rightmost position r where the sources start at $R[r].j' \leq pos$. An RMQ on $R[1..r]$ then finds the position k of the source with the rightmost endpoint $R[k].j''$ in $R[1..r]$. If even $R[k].j'' < pos+m-1$, then no source covers the occurrence and we finish. If, instead, $R[k].j'' \geq pos + m - 1$, then the source $R[k]$ covers the occurrence and we process its corresponding target as a secondary occurrence; in this case we also recurse on the ranges $R[1..k-1]$ and $R[k+1..r]$ that are nonempty. It is easy to see that each valid secondary occurrence is identified in $\mathcal{O}(1)$ time (see Muthukrishnan [41] for an analogous process). In addition, such secondary occurrences, $S[pos'..pos'+m-1]$, must be recursively processed for further secondary occurrences. A similar procedure is described for tracking the secondary occurrences in the LZ77-index [38].

The cost per secondary occurrence reported then amortizes to a predecessor search, $\mathcal{O}(\lg\lg_\omega(n/\gamma))$ time. This cost is also paid for each primary occurrence, which might not yield any secondary occurrence to amortize it. We now prove that this process is sufficient to find all the secondary occurrences.

Lemma 4. *The described algorithm reports every secondary occurrence exactly once.*

Proof. We use induction on the level l of the unmarked block that contains the secondary occurrence. All the secondary occurrences of the last level are correctly reported once, since there are none. Now consider an occurrence $S[pos..pos+m-1]$ inside an unmarked block $S[i..i']$ of level l. This block is the target of a source $S[j'..j'']$ that spans 1 or 2 consecutive marked blocks of level l. Then there is another occurrence $S[pos'..pos'+m-1]$, with $pos' = pos - i + j'$. Note that the algorithm can report $S[pos..pos+m-1]$ only as a copy of $S[pos'..pos'+m-1]$. If $S[pos'..pos'+m-1]$ is primary, then $S[pos..pos+m-1]$ will be reported right after $S[pos'..pos'+m-1]$, because $[pos'..pos'+m-1] \subseteq [j'..j'']$, and the algorithm will map $[j'..j'']$ to $[i..i']$ to discover $S[pos..pos+m-1]$. Otherwise, $S[pos'..pos'+m-1]$ is secondary, and thus it is within one of the marked blocks at level l that overlap the source. Moreover, it is within one of the blocks of level $l+1$ into which those marked blocks are split. Thus, $S[pos'..pos'+m-1]$ is within an unmarked block of level $> l$, which by the inductive hypothesis will be reported exactly once. When $S[pos'..pos'+m-1]$ is reported, the algorithm will also note that $[pos'..pos'+m-1] \subseteq [j'..j'']$ and will find $S[pos..pos+m-1]$ once as well.

If we handle patterns of length $m = 1$ by taking the corresponding attractor positions as the primary occurrences, then there may be secondary occurrences in the last level, but those point directly to primary occurrences (i.e., attractor positions), and therefore the base case of the induction holds too. $\square$

The total search cost with *occ* primary and secondary occurrences is therefore $\mathcal{O}(m(\lg^\epsilon(\gamma\lg(n/\gamma)) + \lg(mn/\gamma)) + occ(\lg^\epsilon(\gamma\lg(n/\gamma)) + \lg\lg_\omega(n/\gamma))) = \mathcal{O}(m\lg n + occ(\lg^\epsilon\gamma + \lg\lg(n/\gamma))) = \mathcal{O}(m\lg n + occ\lg^\epsilon n)$, for any constant $\epsilon > 0$ defined at indexing time (the choice of ϵ affects the constant that accompanies the size $\mathcal{O}(\gamma\lg(n/\gamma))$ of the structure G).

4 Construction

If we allow the index construction to be correct with high probability only, then we can build it in $\mathcal{O}(n + w \lg(n/\gamma) + w\sqrt{\lg w})$ time and $\mathcal{O}(w)$ space (plus read-only access to S), using a Monte Carlo method. Since $w = \mathcal{O}(\gamma \lg(n/\gamma))$ is the number of leaves in the Γ-tree and $\gamma(\lg(n/\gamma))^{\mathcal{O}(1)} \subseteq \mathcal{O}(n)$, the time can be written as $\mathcal{O}(n + w\sqrt{\lg \gamma})$. In order to ensure a correct index, a Las Vegas method requires $\mathcal{O}(n \lg n)$ time in expectation (and w.h.p.) and $\mathcal{O}(n)$ space.

4.1 Building the Γ*-tree*

Given the attractor Γ, we can build the index data structure as follows. At each level l, we create an Aho–Corasick automaton [1] on the unmarked blocks at this level (i.e., those at distance $\geq b_l$ from any attractor), and use it to scan the areas $S[j - b_l + 1..j + b_l - 1]$ around all the attractor elements $j \in \Gamma$ in order to find a proper pointer for each of those unmarked blocks. This takes $\mathcal{O}(n)$ time per level. Since the areas around each of the γ attractor positions are scanned at each level but they have exponentially decreasing lengths, the scanning time adds up to $\mathcal{O}(\gamma\frac{n}{\gamma} + \gamma\frac{n}{2\gamma} + \gamma\frac{n}{4\gamma} + \cdots) = \mathcal{O}(n)$.

As for preprocessing time, each unmarked block is preprocessed only once, and they add up to $\mathcal{O}(n)$ symbols. The preprocessing can thus be done in time $\mathcal{O}(n)$ [19]. To be able to scan in linear time, we can build deterministic dictionaries on the edges outgoing from each node, in time $\mathcal{O}(n(\lg\lg\sigma)^2)$ [46].

In total, we can build the Γ-tree in $\mathcal{O}(n(\lg\lg\sigma)^2)$ deterministic time and $\mathcal{O}(n)$ space.

To reduce the space to $\mathcal{O}(w)$, instead of inserting the unmarked blocks into an Aho–Corasick automaton, we compute their Karp–Rabin fingerprints, store them in a hash table, and scan the areas $S[j-b_l+1..j+b_l-1]$ around attractor elements j. This finds the correct sources for all the unmarked blocks w.h.p. Indeed, if we verify the potential collisions, the result is always correct within $\mathcal{O}(n)$ expected time (further, this time holds w.h.p.).

4.2 Building the Fingerprints

Building the structures for Lemma 1 requires (i) computing the fingerprint of every text prefix ending at block boundaries ($\mathcal{O}(n)$ time and $\mathcal{O}(w)$ space in addition to S), (ii) computing the fingerprint of every explicit block ($\mathcal{O}(w)$ time and $\mathcal{O}(w)$ space starting from the leaves and combining results up to the root), (iii) for each unmarked explicit block B, computing the fingerprint of a string of length at most $|B|$ (i.e., the fingerprint of $B'[i..b_l]$; see case (B) of Lemma 1). Since unmarked blocks do not have children, each text character is seen at most once while computing these fingerprints, which implies that these values can also be computed in $\mathcal{O}(n)$ time and $\mathcal{O}(w)$ space in addition to S.

This process, however, does not include finding a collision-free Karp–Rabin hash function. As a result, the fingerprinting is correct w.h.p. only. We can use the de-randomization procedure of Bille et al. [9], which guarantees to find — in $\mathcal{O}(n \lg n)$ expected time[5] and $\mathcal{O}(n)$ words of space — a Karp–Rabin hash function that is collision-free among substrings of S whose lengths are powers of two. This is sufficient to deterministically check the equality of substrings[6] in the z-fast trie used in the technique [26, Lem. 5.2] that we use to quickly find ranges of pattern suffixes/prefixes (in our Section 3.1).

4.3 Building the Multisets $\mathcal{X}$ and $\mathcal{Y}$

To build the multisets $\mathcal{X}$ and $\mathcal{Y}$ for the primary occurrences, we can build the suffix arrays [40] of S and its reverse, S^{rev}. This requires $\mathcal{O}(n)$ deterministic time and space [31]. Then we can scan those suffix arrays to enumerate $\mathcal{X}$ and $\mathcal{Y}$ in the lexicographic order.

To sort $\mathcal{X}$ and $\mathcal{Y}$ within $\mathcal{O}(w)$ space, we can build instead a sparse suffix tree on the w positions of $\mathcal{X}$ or $\mathcal{Y}$ in S. This can be done in expected time (and w.h.p.) $\mathcal{O}(n\sqrt{\lg w})$ and $\mathcal{O}(w)$ space [27]. If we aim to build the suffix array correctly w.h.p. only, then the time drops to $\mathcal{O}(n)$.

We must then build the z-fast trie [2, Thm. 5] on the sets $\mathcal{X}$ and $\mathcal{Y}$. Since we can use any space in $\mathcal{O}(w)$, we opt for a simpler variant described by Kempa and Kosolobov [34, Lem. 5], which is easier to build. Theirs is a compact trie that stores, for each node v representing the string $v.str$ and with parent node $v.par$: (i) the length $|v.str|$, (ii) a dictionary mapping each character c to the child node v' of v such that $v'.str[|v.str|+1] = c$ (if such a child exists), and (iii) the (non-extended) fingerprint of $v.str[1..k_v]$, where k_v is the *two-fattest number* in the range $[|v.par.str|+1..|v.str|]$, that is, the number in that range whose binary representation has the largest number of trailing zeros. The trie also requires a global "navigating" hash table that maps the $\mathcal{O}(w)$ pairs $(k_v, \hat{\phi}(v.str[1..k_v]))$ to their corresponding node v.

[5] The time is also $\mathcal{O}(n \lg n)$ w.h.p., not only in expectation.
[6] If the length ℓ of the two substrings is not a power of two, then we compare their prefixes and suffixes whose length is the largest power of two smaller than ℓ.

If p prefixes some string in $\mathcal{X}$ (resp. $\mathcal{Y}$) and the fingerprint function $\hat{\phi}$ is collision-free among equal-length text substrings, then their so-called *fat binary search* procedure finds — in time $\mathcal{O}(\lg |p|)$ — the highest node v such that $v.str$ is prefixed by a search pattern p (assuming constant-time computation of the fingerprint of any substring of p, which can be achieved after a linear-time preprocessing of the pattern). The range of strings of $\mathcal{X}$ (resp. $\mathcal{Y}$) is then associated with the node v.

Their search procedure, however, has an anomaly [34, Lem. 5]: in some cases it might return a child of v instead of v. We can fix this problem (and always return the correct v) by also storing, for every internal trie node v, the fingerprint of $v.str$ and the exit character $v.str[|v.par.str| + 1]$. Then, we know that the procedure returned v if and only if $|v.par.str| < |p|$, the fingerprint of $v.par.str$ matches that of $p[1..|v.par.str|]$ and $v.str[|v.par.str| + 1] = p[|v.par.str| + 1]$. If these conditions do not hold, we know that the procedure returned a child of v and we fix the answer by moving to the parent of the returned node. [7]

If p does not prefix any string in $\mathcal{X}$ (resp. $\mathcal{Y}$), or if $\hat{\phi}$ is not collision-free, the search (with our without our fix) returns an arbitrary range. We recall that, to cope with the first condition (which gives the z-fast trie search the name *weak* prefix search), Lemma 3 adds a deterministic check that allows discarding the incorrect ranges returned by the procedure. The second condition, on the other hand, w.h.p. does not fail. Still, we recall that we can ensure that our function is collision-free by checking, at construction time, for collisions among substrings whose lengths are powers of two, as described in Section 4.2.

The construction of this z-fast trie starts from the sparse suffix tree of $\mathcal{X}$ (or $\mathcal{Y}$), since the topology is the same. The dictionaries giving constant-time access to the nodes' children can be built correctly w.h.p. in time $\mathcal{O}(w)$, or be made perfect in expected time (and w.h.p.) $\mathcal{O}(w)$ [51]. Alternatively, one can use deterministic dictionaries, which can be built in worst-case time $\mathcal{O}(w(\lg\lg\sigma)^2)$ [46]. In all cases, the construction space is $\mathcal{O}(w)$. Similarly, the navigating hash can be built correctly w.h.p. in time $\mathcal{O}(w)$, or be made perfect in expected time (and w.h.p.) $\mathcal{O}(w)$. We could also represent the navigating hash using deterministic dictionaries built in worst-case time $\mathcal{O}(w(\lg\lg w)^2)$ [46] and space $\mathcal{O}(w)$. The $\mathcal{O}(w)$ fingerprints $\hat{\phi}(v.str)$ and the hashes of the pairs $(k_v, \hat{\phi}(v.str[1..k_v]))$ can be computed in total time $\mathcal{O}(w \lg(n/\gamma))$ by using Lemma 1 to compute the Karp–Rabin fingerprints.

[7] This fix works in general. In their particular procedure, we do not need to check that $v.str[|v.par.str|+1] = p[|v.par.str|+1]$ (nor to store the exit character). Alternatively, we could directly fix their procedure by similarly storing the fingerprint of $v.str$ ($\mathsf{hash}(v.str)$, in their article) and changing line 6 of their extended version [34] to "**if** $v.len < |pat|$ **and** $\mathsf{hash}(v.str) = \mathsf{hash}(pat[1..v.len])$ **and** $v.map(pat[v.len + 1]) \neq$ **nil then** $v \leftarrow v.map(pat[v.len + 1])$;".

A variant of the grid data structure G [6], with the same space and time performance, can be built in $\mathcal{O}(w\sqrt{\lg w})$ deterministic time, and $\mathcal{O}(w)$ space. The arrays T and R can be built in $\mathcal{O}(w)$ space and $\mathcal{O}(w \lg\lg w)$ deterministic time [28].

The RMQ structure on R requires $\mathcal{O}(w)$ deterministic construction time and $\mathcal{O}(w)$ bits [21]. The predecessor data structure [45], however, requires perfect hashing. This can be built in $\mathcal{O}(w)$ space and $\mathcal{O}(w(\lg\lg w)^2)$ deterministic time [46], or in $\mathcal{O}(w)$ expected time (also holding w.h.p.).

5 Conclusions

We have introduced the first *universal* self-index for repetitive text collections. The index is based on a recent result [35] that unifies a large number of dictionary compression methods into the single concept of *string attractor*. For each compression method based on Lempel–Ziv, grammars, run-compressed BWT, collage systems, macro schemes, etc., it is easy to identify an attractor set of the same asymptotic size obtained by the compression method, say γ. Thus, our construction automatically yields a self-index for each of those compression methods, within $\mathcal{O}(\gamma \lg(n/\gamma))$ space. No structure is known of size $o(\gamma \lg(n/\gamma))$ able to efficiently extract a substring from the compressed text (say, within $\mathcal{O}(\text{polylog}\, n)$ time per symbol), and thus $\mathcal{O}(\gamma \lg(n/\gamma))$ is the minimum space we could hope for an efficient self-index with the current state of the art.

Indeed, no self-index of size $o(\gamma \lg(n/\gamma))$ is known: the smallest ones use $\mathcal{O}(z)$ [38] and $\mathcal{O}(r)$ [26] space, respectively, yet there are text families where $z, r = \Omega(b^* \lg n) = \Omega(\gamma^* \lg n)$ [25] (where b^* and γ^* denote the smallest macro scheme and attractor sizes, respectively). The most time-efficient self-indexes use $\Theta(r \lg(n/r))$ and $\Omega(z \lg(n/z))$ space, which is asymptotically equal or larger than ours. The search time of our index, $\mathcal{O}(m \lg n + occ \lg^{\epsilon} n)$ for any constant $\epsilon > 0$, is close to that of the fastest of those self-indexes, which were developed for a specific compression format (see [26, Table 1]). Moreover, our construction provides a self-index for compression methods for which no such structure existed before, such as collage systems and macro schemes. Those can provide smaller attractor sets than the ones derived from the more popular compression methods.

We can improve the search time of our index by using slightly more space. Our current bottleneck in the per-occurrence query time is the grid data structure G, which uses $\mathcal{O}(w)$ space and returns each occurrence in time $\mathcal{O}(\lg^\epsilon w)$. Instead, a grid structure [12] using $\mathcal{O}(w \lg\lg w) = \mathcal{O}(\gamma \lg(n/\gamma) \lg\lg n)$ space obtains the occ_p primary occurrences in time $\mathcal{O}((occ_p + 1) \lg\lg w)$. This slightly larger version of our index can then search in time $\mathcal{O}(m \lg n + occ \lg\lg n)$. This complexity is close to that of some larger indexes in the literature for repetitive string collections (see [26, Table 1]).

A number of avenues for future work are open, including supporting more complex pattern matching, handling dynamic collections of texts, supporting document retrieval, and implementing a practical version of the index. Any advance in this direction will then translate into all of the existing indexes for repetitive text collections.

Acknowledgements

We thank the reviewers for their outstanding job in improving our presentation (and even our bounds).

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, pages 427–438, 2010.

[3] D. Belazzougui and F. Cunial. Fast label extraction in the CDAWG. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 10508, pages 161–175, 2017.

[4] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 26–39, 2015.

[5] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S. J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. 25th Data Compression Conference (DCC)*, pages 83–92, 2015.

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

[7] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, LNCS 9294, pages 142–154, 2015.

[8] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theoretical Computer Science*, 713:66–77, 2018.

[9] P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 65–76, 2015.

[10] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

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

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

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

[14] A. R. Christiansen and M. B. Ettienne. Compressed indexing with signature grammars. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 10807, pages 331–345, 2018.

[15] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[16] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[17] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192, 2012.

[18] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1264, pages 116–129. Springer, 1997.

[19] S. Dori and G. M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Information Processing Letters*, 98(2):66–72, 2006.

[20] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[21] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[22] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[23] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, LNCS 7183, pages 240–251, 2012.

[24] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 8392, pages 731–742, 2014.

[25] T. Gagie, G. Navarro, and N. Prezza. On the approximation ratio of Lempel-Ziv parsing. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 10807, pages 490–503, 2018.

[26] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.

[27] P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 425–439, 2017.

[28] Y. Han. Deterministic sorting in $O(n \lg \lg n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[29] A. Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[30] A. Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[31] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[32] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

[33] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[34] D. Kempa and D. Kosolobov. LZ-End parsing in compressed space. In *Proc. 27th Data Compression Conference (DCC)*, pages 350–359, 2017. Extended version in `https://arxiv.org/pdf/1611.01769.pdf`.

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

[36] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

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

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

[39] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

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

[41] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

[42] G. Navarro. A self-index on Block Trees. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 10508, pages 278–289, 2017.

[43] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. In *Proc. Prague Stringology Conference (PSC)*, pages 158–170, 2016.

[44] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 72:1–72:15, 2016.

[45] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[46] M. Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 84–95, 2008.

[47] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[48] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(24):416–430, 2005.

[49] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[50] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[51] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.