Block Tree based Universal Self-Index for Repetitive Text Collections

Pablo Vicente Helguero Vargas
Department of Computer Science
University of Chile
Santiago, Chile
pablo.helguero@ug.uchile.cl

1 Introduction

The rise in the amount of data we aim to handle [45] and the fact that most of today's fastest-growing data is highly repetitive, has triggered recent research on exploiting repetitiveness to enable space reductions of orders of magnitude [21]. Being able to manipulate the text within compressed space, with a compression related to its repetitiveness has a critical importance in many areas of study such as Bioinformatics, Information Retrieval, Data Mining, among others. Typical definitions of repetitiveness either look at the string's composition in terms of distinct k-mers or rely on the output of compressors [47, 48].

One effective compression strategy is to build a context-free grammar that generates only the string. Another strategy is that of replacing repetitions, which point to other locations in the string. The most powerful and general scheme falling into this category is called Pointer Macro Scheme [46]. Other effective techniques to compress repetitive strings include the run-length Burrows-Wheeler transform [7] (RLBWT) and the Compact Directed acyclic Word Graph [6, 16] (CDAWG).

Manipulating the text data in compressed form requires not only compression, but also the ability to access the text directly in compressed form and of performing searches on the compressed text. A self-index on a string S is a data structure that offers direct access to any substring of S, and at the same time supports indexed queries such as counting and locating pattern occurrences in S. Unfortunately, classic self-indexes that work extremely well on standard data sets 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 the logarithm of the input's length [18, 28, 34].

For that reason, recent studies focus on self-indexing based on dictionary compressors such as the Lempel-Ziv factorization (LZ77) [29], the run-length encoded Burrows-Wheeler Transform (RLBWT) [7] and context-free grammars (CFGs) [26], just to name the most popular ones. These schemes allow 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 highly repetitive data sets.

Kempa and Prezza [25] show that the above compressibility measures are dominated by a new lower bound called string attractor. An attractor Γ is a set of positions in S such that any substring of S has an occurrence covering a position in Γ. The size γ of the smallest attractor asymptotically lower bounds all the repetitiveness measures of the techniques described above.

Navarro and Prezza [38] proposed the first index based on string attractors, which is called "universal" because it builds on this stronger compressibility measure and thus lower-bounds the space of most other indexes. It was not implemented, and left open the important challenge that finding the smallest attractor is NP-complete [25].

This proposal focuses on building a practical universal self-index for highly repetitive texts. It builds on the work of Navarro and Prezza, whose data structure is a variant of Block Trees [27]. Instead of building on attractors, however, we will modify the original Block Tree structure so that its size is bounded by an even stricter measure, $\delta \leq \gamma$ [11], which can be computed in linear time and enables smaller Block Trees, as demonstrated in the theoretical work of Kociumaka et al. [27]. The thesis must afford various challenges towards obtaining a practical implementation of the idea, which will then be compared with other state-of-the-art indexes. The rest of the document presents in more detail the central concepts to understand the problem.

2 Related Work

Let $T[1..n]$ be a text and let us define some measures of repetitiveness that we use for the rest of the document. Lempel and Ziv [29] proposed a method that parses the text T into *phrases* and we define the Lempel-Ziv measure of T as the number z of phrases into which T is parsed. Kieffer and Yang [26] introduced a compression technique based on context-free grammars where we find a grammar that generates only the text T, then our measure of repetitiveness is the size g of the smallest grammar that generates only T. Burrows and Wheeler [7] designed a reversible transformation to make strings easier to compress, they obtain a permutation T^{bwt} of T, and we define r as the number of equal-symbol runs in T^{bwt}. A Compact Directed Acyclic Word Graph (CDAWG) [6] is obtained by collapsing all the leaves of a suffix tree and minimizing it as an automaton, then the size e of the CDAWG of T mesured in terms of nodes plus edges is a repetitiveness measure. Experimental results [31, 28, 2, 12] suggest that in typical repetitive texts it holds $z < r \approx g \ll e$.

For highly repetitive texts, one hopes to have a compressed index, which replaces the text T with a compressed version that nonetheless can efficiently extract any substring $T[i..j]$ without having to fully decompress it. Indexes that, implicitly contain a replacement of T, are called self-indexes, and those with $O(z)$ space require up to $O(z)$ time per extracted character [28]. Good extraction times, typically $O(\log n)$, are instead obtained with $O(g)$ [13, 14, 36, 19],$O(z \log \frac{n}{z})$ [42, 9, 43] or $O(e)$ [1, 2, 4] space. A lower bound for grammar-based representations [50] shows that $\Omega((\log n)^{1-\varepsilon} / \log g)$ time, for any constant $\varepsilon > 0$, is needed to access one random position within $O(poly(g))$ space.

The first research on indexing and searching repetitive collections were made by different authors [30, 44, 31, 32]. Their index, Run-Length FM-Index (RLFM-index) uses $O(r)$ words, and can count the number of occurrences of a pattern $P[1..m]$ in time $O(m \log n)$ and even less. However, they are unable to locate where those positions are in T unless they add a set of samples that require $\Theta(n/s)$ words to offer $O(s \log n)$ time to locate each occurrence, where s is a space/time trade-off parameter. Kreft and Navarro [28] introduced a self-index based on LZ77 compression, extremely space-efficient on highly repetitive text collections [12]. It uses $O(z)$ space and finds all the *occ* occurrences of a pattern in time $O((m^2 h + (m + occ) \log z) \log (n/z))$, where $h \leq z$ is the maximum number of times a symbol is copied.

Based on a representation of a context -free grammar [15] with g symbols, size G and height h, all the occurrences occ of a pattern of length m are found in time $O((m^2/\varepsilon)\log\log n+(m+occ)(1/\varepsilon+\log g/\log\log g))$ using a structure built in $O(n+G\log G)$ time and $O(n\log n)$ bits of space, for any constant $\varepsilon > 0$. Some of the fastest indexes known are based on the Run-Length BWT and on the Compact Directed Acyclic Word Graph [6, 16].

Various indexes based on combinations of LZ77, CFGs, and Run-Length BWTs have also been proposed [19, 20, 2, 39, 5, 10]. Some of their best results are $O(z\log\frac{n}{z}+z\log\log z)$ space with $O(m+occ(\log\log n+\log^{\varepsilon} z))$ query time [10], and $O(z\log\frac{n}{z})$ space with either $O(m\log m+occ\log\log n)$ [20] or $O(m+\log^{\varepsilon} z+occ(\log\log n+\log^{\varepsilon} z))$ [10] query time where occ is the number of occurrences of a pattern of length m and for any constant $\varepsilon > 0$. Many of the above indexes do not have a known implementation, though Claude et al. [15] compare different state-of-the-art indexes on repetitive collections and a grammar-based self-index.

Kempa and Prezza [25] proposed a direct measure of repetitiveness on the text T instead of the result of a specific compression method. They unified the existing measures into a more abstract characterization of the string. An attractor of T is a set Γ of positions in T such that any substring $T[i..j]$ must have a copy including an elemento of Γ. The substrings of a repetitive string should be covered with smaller attractors. The measure is then γ, the smallest size of an attractor Γ of T. In general, it is NP-complete to find the smallest attractor size for T [25], but in exchange they show that $\gamma = O(min(z,r,g))$.

Our last measure of repetitiveness for a text T, δ, is built on top of the concept of *string complexity*, that is, the number $T(k)$ of distinct substrings of length k. Raskhodnikova et al. [41] defines $\delta = max\{T(k)/k : k \in [1..n]\}$. It is not hard to see that $\delta \leq \gamma$ for every text T [11]: Since every substring of length k in T has no copy including some of its γ attractor elements, there can be only $k\gamma$ distinct substrings, that is, $T(k) \leq k\gamma$ for all k. Christiansen et al. [11] also show how δ can be computed in linear time but it needs a suffix tree [51, 33, 49].

Block Trees [3] are in principle built knowing the size z of the text $T[1..n]$. Built with a parameter $\tau = O(1)$, they provide a way to access any $T[i]$ in time $O(\log(n/z))$ with a data structure of size $O(z\log(n/z))$, which is also the best asymptotic space obtained with grammar compressors [9, 43, 23, 24, 42] but they can be asymptotically faster.

The block tree is of height $\log_{\tau}(n/z)$. The root has z children, $u_1,...,u_z$, which divide T into blocks of length n/z, $T = T_{u_1}...T_{u_z}$. Each such node $v = u_j$ has τ children, $v_1,...,v_{\tau}$, which divide its block T_v into equal parts, $T_v = T_{v_1}...T_{v_{\tau}}$. The nodes v_i have, in turn, τ children that subdivide their block, and so on. After slightly less than $\log_{\tau}(n/z)$ levels, the blocks are of length $\log_{\sigma} n$ where σ is the size of the alphabet of T, and can be stored explicitly using $\log n$ bits, that is, in constant space.

Some of the nodes v can be removed because their block T_v appears earlier in T. Every consecutive pair of nodes v_1, v_2 where the concatenation $T_{v_1}T_{v_2}$ does not appear earlier is *marked*. After this, every unmarked node v has an earlier occurrence, so instead of creating its τ children, we replace v by a leftward pointer to the first occurrence of T_v in T. This first occurrence spans in general two consecutive nodes v_1, v_2 at the same level of v, and these exist and are marked by construction. We then make v a leaf pointing to v_1, v_2, also recording the offset where T_v occurs inside $T_{v_1}T_{v_2}$.

Kempa and Prezza [25] build a similar structure on top of an attractor Γ of T, of minimal size $\gamma \leq z$. Their structure uses $O(\gamma \log(n/\gamma))$ space and extracts any $T[i..j]$ in time $O((1+(j-i)/\log_\sigma n)\log(n/\gamma))$. Navarro and Prezza [38] describe the so-called Γ-tree, which is more similar to a block tree and more suitable for indexing. This Γ-tree takes $O(\gamma \log(n/\gamma))$ words of spaces, supports locating the *occ* occurrences of any pattern of length m in $O(m \log n + occ \log^\varepsilon n)$ time, for any constant $\varepsilon > 0$, and can retrieve any $T[i]$ in time $O(\log(n/\gamma))$. So it matches the substring extraction time of Kempa and Prezza [25].

Recently, Kociumaka et al. [27] showed that the original block tree is easily turned to use $O(\delta \log n/\delta)$ space. The only change needed is to start with δ top-level blocks, it can be seen that there are only $O(\delta)$ marked blocks per level. The tree height is $O(\log(n/\delta))$, higher than the block tree. However, they obtain that $\log(n/\delta) = O(\log(n/z))$, and therefore the difference in query times is not asymptotically relevant. Cáceres [8] developed an implementation of block trees. This structure promises to be faithful to its theoretical description and slightly larger than grammar-based structures but outperforms them in time. We will prove that this structure uses $O(\delta \log n/\delta)$ space to implement a practical self-index on top of it.

3 Problem Statement

There is a theoretical proposal of a universal compressed self-index [38], a modification of a block tree structure [27], both of them with theoretically very low usage of space and fast query time, and also an implementation of block trees [8] which we will prove that uses $O(\delta \log n/\delta)$ space. So the main aim of this work is to propose a practical implementation of the universal compressed self-index described by Navarro and Prezza [38] using the block tree variation made by Kociumaka et al. [27] through the implementation of [8].

4 Research Questions

Is it practical to implement the universal compressed self-index proposed by Navarro and Prezza [38] using a block tree in δ-bounded space structure [27]? Is this implementation competitive in time locating occurrences and extracting substrings compared to the state-of-the-art indexes? How big is the space used by this index compared to those mentioned above? In which kind of repetitive text is competitive in terms of time and space?

5 Hypothesis

A universal compressed self-index using a block tree in δ-bounded space structure will be faster in time locating occurrences, extracting substrings, but will use less space in highly repetitive text collections and more space in less repetitive text collections than all other state-of-the-art methods.

6 Main Goal

6.1 General Objectives

The general objective of this thesis is to implement the universal self-index for highly repetitive text collections described by Navarro and Prezza [38]. We will modify its structure, based on the ideas of Kociumaka et al. [27], using a block tree variant implemented by Cáceres [8]. The proposed index will be evaluated with different state-of-the-art indexes as done in previous works [15, 8] focusing on the location of occurrences, substring extraction, and space used.

6.2 Specific Objectives

– Implement the universal compressed self-index for highly repetitive text collections described by Navarro and Prezza [38], using the block tree in δ-bounded space described by Kociumaka et al. [27]. This will based on the block tree implementation made by Cáceres [8].

– Compare the proposed self-index with state-of-the-art indexes in time of location of occurrences and substring extraction.

– Analyze the space usage of the proposed index in relation to the state-of-the-art indexes.

7 Methodology

7.1 Research

The first step in the research is to do a short review of the state-of-the-art indexes for highly repetitive text collections, an analysis of complexities, space used, and time of queries. This will additionally require a deeper analysis of strings attractors, string complexity, and block trees. On top of that, to achieve a block-tree-based representation of the universal compressed self-index and various of the components of these kinds of indexes, algorithm engineering will be necessary for the implementation of the most theoretical parts of the structures. Also, we will study if it is possible to optimize these structures for the particular case of the Block Tree.

7.2 A universal self-index based on block trees library

It will be useful to implement a library of the universal compressed self-index based on block trees to perform experiments, do benchmarking, and leave it public so it can be tested by anyone. It will be based on the block tree construction implemented by Cáceres [8]. The proposed self-index will be developed in C++ because of the maturity of the API, familiarity with the language, and because the said block tree implementation is written in this language.

7.3 Experimentation

The experimental evaluation will be carried out using the environment provided by Ferragina and Navarro [17] as exhibited in similar works [15, 8]. We will compare the proposed implementation with the available indexes used in the previously mentioned works, which are some of the state-of-the-art indexes. These studies show six real repetitive collections that we use. Three of these collections contain DNA sequences extracted from different sources, a collection formed from all versions of the articles in English of Albert Einstein taken from Wikipedia. Finally, collections of *kernels* and *coreutils* formed by all versions 5.x of the Coreutils package and all 1.0.x and 1.1.x versions of the Linux Kernel.

The experimentation will evaluate the space-time trade-offs obtained for locating patterns of different lengths over the indexes on all collections described above. We will show how the location time evolves with different pattern lengths and the time per extracted symbol of the different indexes and collections when extracting different consecutive text symbols.

8 Expected results

- A new self-index that speeds up the extraction and search of occurrences of a pattern in highly repetitive text collections.
- An open source library to facilitate the integration on top of other compressors.
- A significant speed up in performance that outperforms the state-of-the-art indexes in time.
- As suggested by other similar works [8], a slightly larger representation in space is expected for this index.

References

[1] Djamal Belazzougui & Fabio Cunial (2017): *Fast label extraction in the CDAWG.* In: *International Symposium on String Processing and Information Retrieval*, Springer, pp. 161–175.

[2] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza & Mathieu Raffinot (2015): *Composite repetition-aware data structures.* In: *Annual Symposium on Combinatorial Pattern Matching*, Springer, pp. 26–39.

[3] Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi & Yasuo Tabei (2015): *Queries on LZ-bounded encodings.* In: *2015 Data Compression Conference*, IEEE, pp. 83–92.

[4] Djamal Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini & Jouni Sirén (2014): *Relative FM-indexes.* In: *International Symposium on String Processing and Information Retrieval*, Springer, pp. 52–64.

[5] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz & Hjalte Wedel Vildhøj (2018): *Time–space trade-offs for lempel–Ziv compressed indexing.* *Theoretical Computer Science* 713, pp. 66–77.

[6] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell & Andrzej Ehrenfeucht (1987): *Complete inverted files for efficient text retrieval and analysis.* *Journal of the ACM (JACM)* 34(3), pp. 578–595.

[7] Michael Burrows & David J Wheeler (1994): *A block-sorting lossless data compression algorithm.*

[8] Manuel Ariel Cáceres Reyes (2019): *Compressed suffix trees for repetitive collections based on block trees.*

[9] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai & Abhi Shelat (2005): *The smallest grammar problem.* *IEEE Transactions on Information Theory* 51(7), pp. 2554–2576.

[10] Anders Roy Christiansen & Mikko Berggren Ettienne (2018): *Compressed indexing with signature grammars.* In: *Latin American Symposium on Theoretical Informatics*, Springer, pp. 331–345.

[11] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro & Nicola Prezza (2018): *Optimal-time dictionary-compressed indexes.* *arXiv preprint arXiv:1811.12779.*

[12] Francisco Claude, Antonio Fariña, Miguel A Martínez-Prieto & Gonzalo Navarro (2016): *Universal indexes for highly repetitive document collections.* *Information Systems* 61, pp. 1–23.

[13] Francisco Claude & Gonzalo Navarro (2011): *Self-indexed grammar-based compression*. *Fundamenta Informaticae* 111(3), pp. 313–337.

[14] Francisco Claude & Gonzalo Navarro (2012): *Improved grammar-based compressed indexes*. In: *International Symposium on String Processing and Information Retrieval*, Springer, pp. 180–192.

[15] Francisco Claude, Gonzalo Navarro & Alejandro Pacheco (2020): *Grammar-Compressed Indexes with Logarithmic Search Time*. *arXiv preprint arXiv:2004.01032*.

[16] Maxime Crochemore & Renaud Vérin (1997): *Direct construction of compact directed acyclic word graphs*. In: *Annual Symposium on Combinatorial Pattern Matching*, Springer, pp. 116–129.

[17] P Ferragina & G Navarro: *Pizza&Chili Corpus—Compressed indexes and their testbeds, 2005*.

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

[19] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich & Simon J Puglisi (2012): *A faster grammar-based self-index*. In: *International Conference on Language and Automata Theory and Applications*, Springer, pp. 240–251.

[20] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich & Simon J Puglisi (2014): *LZ77-based self-indexing with faster pattern matching*. In: *Latin American Symposium on Theoretical Informatics*, Springer, pp. 731–742.

[21] Travis Gagie, Gonzalo Navarro & Nicola Prezza (2020): *Fully functional suffix trees and optimal text searching in bwt-runs bounded space*. *Journal of the ACM (JACM)* 67(1), pp. 1–54.

[22] Simon Gog, J Bader, T Beller & M Petri (2014): *Sdsl-succinct data structure library*.

[23] Artur Jeż (2015): *Approximation of grammar-based compression via recompression*. *Theoretical Computer Science* 592, pp. 115–134.

[24] Artur Jeż (2016): *A really simple approximation of smallest grammar*. *Theoretical Computer Science* 616, pp. 141–150.

[25] Dominik Kempa & Nicola Prezza (2018): *At the roots of dictionary compression: String attractors*. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 827–840.

[26] John C Kieffer & En-Hui Yang (2000): *Grammar-based codes: a new class of universal lossless source codes*. *IEEE Transactions on Information Theory* 46(3), pp. 737–754.

[27] Tomasz Kociumaka, Gonzalo Navarro & Nicola Prezza (2019): *Towards a definitive measure of repetitiveness*. *arXiv preprint arXiv:1910.02151*.

[28] Sebastian Kreft & Gonzalo Navarro (2013): *On compressing and indexing repetitive sequences*. *Theoretical Computer Science* 483, pp. 115–133.

[29] Abraham Lempel & Jacob Ziv (1976): *On the complexity of finite sequences*. *IEEE Transactions on information theory* 22(1), pp. 75–81.

[30] Veli Mäkinen & Gonzalo Navarro (2005): *Succinct suffix arrays based on run-length encoding*. In: *Annual Symposium on Combinatorial Pattern Matching*, Springer, pp. 45–56.

[31] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén & Niko Välimäki (2009): *Storage and retrieval of individual genomes*. In: *Annual International Conference on Research in Computational Molecular Biology*, Springer, pp. 121–137.

[32] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén & Niko Välimäki (2010): *Storage and retrieval of highly repetitive sequence collections*. *Journal of Computational Biology* 17(3), pp. 281–308.

[33] Edward M McCreight (1976): *A space-economical suffix tree construction algorithm*. *Journal of the ACM (JACM)* 23(2), pp. 262–272.

[34] Gonzalo Navarro (2016): *Compact data structures: A practical approach*. Cambridge University Press.

[35] Gonzalo Navarro (2017): *A self-index on block trees*. In: *International Symposium on String Processing and Information Retrieval*, Springer, pp. 278–289.

[36] Gonzalo Navarro (2019): *Document listing on repetitive collections with guaranteed performance*. Theoretical Computer Science 772, pp. 58–72.

[37] Gonzalo Navarro (2020): *Indexing Highly Repetitive String Collections*. arXiv preprint arXiv:2004.02781.

[38] Gonzalo Navarro & Nicola Prezza (2019): *Universal compressed text indexing*. Theoretical Computer Science 762, pp. 41–50.

[39] Takaaki Nishimoto, I Tomohiro, Shunsuke Inenaga, Hideo Bannai & Masayuki Takeda (2015): *Dynamic index, LZ factorization, and LCE queries in compressed space*. arXiv preprint arXiv:1504.06954.

[40] Nicola Prezza (2018): *Optimal rank and select queries on dictionary-compressed text*. arXiv preprint arXiv:1811.01209.

[41] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld & Adam Smith (2013): *Sublinear algorithms for approximating string compressibility*. Algorithmica 65(3), pp. 685–709.

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

[43] Hiroshi Sakamoto (2005): *A fully linear-time approximation algorithm for grammar-based compression*. Journal of Discrete Algorithms 3(2-4), pp. 416–430.

[44] Jouni Sirén, Niko Välimäki, Veli Mäkinen & Gonzalo Navarro (2008): *Run-length compressed indexes are superior for highly repetitive sequence collections*. In: International Symposium on String Processing and Information Retrieval, Springer, pp. 164–175.

[45] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha & Gene E Robinson (2015): *Big data: astronomical or genomical?* PLoS biology 13(7), p. e1002195.

[46] James A Storer & Thomas G Szymanski (1982): *Data compression via textual substitution*. Journal of the ACM (JACM) 29(4), pp. 928–951.

[47] Edward N Trifonov (1990): *Making sense of the human genome*. Structure and methods: proceedings of the Sixth Conversation in the Discipline Biomolecular Stereodynamics held at the State University of New York at Albany, June 6-10, 1989/edited by RH Sarma & MH Sarma.

[48] Olga G Troyanskaya, Ora Arbell, Yair Koren, Gad M Landau & Alexander Bolshoy (2002): *Sequence complexity profiles of prokaryotic genomic sequences: A fast algorithm for calculating linguistic complexity*. Bioinformatics 18(5), pp. 679–688.

[49] Esko Ukkonen (1995): *On-line construction of suffix trees*. Algorithmica 14(3), pp. 249–260.

[50] Elad Verbin & Wei Yu (2013): *Data structure lower bounds on random access to grammar-compressed strings*. In: Annual Symposium on Combinatorial Pattern Matching, Springer, pp. 247–258.

[51] Peter Weiner (1973): *Linear pattern matching algorithms*. In: 14th Annual Symposium on Switching and Automata Theory (swat 1973), IEEE, pp. 1–11.