Compressed Suffix Tree for Repetitive Collections based on Block Trees

November 2017

1 Introduction

The amount of data is in constant growth as well as the need to process them efficiently. Sometimes, complex queries over these data are required and precomputed data structures to support these queries efficiently emerge naturally.

One of the most appreciated data structures in text collections (where we are working with n symbols over an alphabet of finite size σ) are Suffix Trees [41] [26] [40], which are capable of solving many problems in stringology [3] [17] such as searching for a pattern, finding the longest repeated substring or the longest common substring between two texts.

However, a serious problem about these trees is their space usage. From an Information Theory standpoint, classical representations use $\Theta(n \log n)$ bits whereas information contained in the text is, in the worst case, $n \log \sigma$ bits, becoming critical for large text collections. From a practical point of view, even carefully engineered implementations [20] require at least 10 bytes per symbol, something that some applications cannot afford because that would mean to use slower levels of memory, making the algorithms over the Suffix Tree orders of magnitude slower.

One field of application where these worries arise is bioinformatics. In bioinformatics there are large DNA sequences on which different types of complex analysis are executed and the use of appropriate data structures is mandatory, being the Suffix Tree probably the most important one. DNA sequences range over 4 different nucleobases represented with 2 bits each ($\log 4 = 2$) while the Suffix Tree uses more than 10 bytes = 80 bits per character, meaning 4000% of the text size. A human genome uses approximately $715MB$ [28], whereas its Suffix Tree requires about $30GB$. This requires machines with great amounts of main memory or resorting to the slower secondary memory. The scene becomes worse when considering the DNA analysis of a group of individuals.

Computer scientists have considered different approaches to deal with this problem. One line of research is to build secondary memory efficient implementations of Suffix Trees [7] [9], however most processing of these trees requires traversing them across arbitrary access paths, bringing the problem of poor locality of reference (which is particularly relevant on secondary memory).

Another approach consists on using the Suffix Array [25] instead, which decreases space usage (4 bytes per character) but loses some important functionalities, for example, the navigational suffix links.

Trying to recover these functionalities [2] with extra information raises space usage again ($\sim$ 6 bytes per character).

A promising line of research is the construction of compact representations of Suffix Trees, named Compressed Suffix Trees, whose objectives are not just achieving the space usage of the worst case representation of the text ($n \log \sigma$ bits) but of its information content (text entropy). An amazing theoretical achievement in this line was the proposal of a compact representation using space proportional to the text size that keeps the full functionality of the tree [38]. Practical implementations based on these ideas face a well known time/space trade-off, some of them achieving as little as $\sim$ 5 bits per character (less than 1 byte) [36] but sacrificing performance.

Despite the progress done, these implementations are not enough to process huge amounts of data, due to their space usage or time complexity. Fortunately, many of the longest text collections are highly repetitive; for example DNA sequences of multiple people differ in less than 0.5% of their content [39]. This repetitiveness is not well captured with statistical based compression methods [19], on which most of the Compressed Suffix Trees are based. A better option to capture this repetitiveness are Lempel-Ziv [23] and Grammar [18] based compression techniques [28], but only recently these kinds of indexes are being adapted to deal with the complex queries needed to implement Compressed Suffix Trees [14] [4].

This work will be dedicated to adapt and implement Block Trees [6] to use them in new implementations of Compressed Suffix Trees. The Block Tree is a relatively new Lempel-Ziv based index that has shown good results in terms of space and access time on highly repetitive text collections [6]. This implementation will be tested on repetitive text collections and compared against other implementations aware of this repetitiveness [1] [31].

2 State of the Art

Compressed Suffix Tree (CST)

The first fully-functional CST (supporting a defined standard set of operations [29, Section 11.5.2]) is due to Sadakane [38]. It is composed of three elements: a Compressed Suffix Array (CSA) [30], the topology of the Suffix Tree represented with balanced parentheses [27], and a compressed version of the Longest Common Prefix (LCP) Array [25]. The version of Sadakane uses $6n + o(n)$ bits on top of the CSA. Improvements on this idea have been made but do not get rid of the $\Theta(n)$ extra bits. A recent implementation [34] requires about 10 bits per character and operates in the order of microseconds.

Another idea due to Russo et. al [36] achieves $o(n)$ bits on top of the CSA. The idea consists on sampling a particular set of nodes of the Suffix Tree, exploiting their relation with suffix links and using the longest common ancestor (lca) operation. An implementation of this idea [37] uses as little as 4 bits per character but decreases time performance to milliseconds per operation.

Finally, a third approach [10] also obtains $o(n)$ on top of CSA. They get rid of the tree topology and express the tree operations on Suffix Array intervals, using previous/next smaller value (psv/nsv) and range minimum query (rmq) operations on the compressed LCP Array. Abeliuk et. al [1] implemented

this idea with the help of Range min-Max Trees [32], obtaining about 8 bits per character and getting a time performance of hundreds of microseconds, an interesting time/space trade-off between the other two options.

Compressed Suffix Tree for Repetitive Collections

Abeliuk et. al [1] also presented the first CST for repetitive collections. They used RLCSA [24], a repetition-aware CSA, and used the parsing tree obtained by the Re-Pair [22] grammar compression of the differential LCP. They tested in repetitive text collections from the *PizzaChilli*[‡] platform, on which the structure used a little less than 2 bits per character. This reaches even 1 bit per character on the most repetitive texts. A problem with this implementation is time, as it operates in the order of milliseconds.

Another implementation, by Navarro and Ordoñez [31], uses a Re-Pair grammar-compressed representation of the tree topology, which they call Grammar-Compressed Topology (GCT). Since they do not need psv/nsv/rmq operations on LCP, they use bitvector H, a particular representation of the LCP Array proposed by Sadakane [38] and adapted to repetitive collections [10] [1]. Their implementation was tested on repetitive text collections from the *PizzaChilli* platform, and obtained slightly higher space usage than Abeliuk et. al [1] (about 2 bits per character in repetitive collections). Yet, this outperformed their competitor in time by three orders of magnitude: their structure operates in the order of microseconds, becoming much closer to the times of general-purpose CSTs.

It is worth mentioning very recent work made by Farruggia et. al [8], who used ideas from Relative Lempel-Ziv compression [21] to compress the Suffix Trees of individual sequences relative to the Suffix Tree of a reference sequence. They showed to be time- and space-competitive against the CSTs mentioned, however the output of this compression are various Suffix Trees (one per text) and not one over the entire collection, which loses some important functionalities. Another recent publication by Gagie et. al [14] presented a CST which uses Run-Length Context-Free Grammars [33] on LCP and Suffix arrays. Finally, Belazzougui and Cunial [4] introduced a representation of the Suffix Tree based on a Compact Direct Acyclic Word Graph [5].

Block Trees

Gagie et. al [12], [13] presented Block Graphs, a Directed Acyclic Graph that covers a text dividing it into 3 equal-sized overlapping blocks. Each block is recursively divided until a block represents a substring previously seen in the text, in which case the block is replaced by a pointer to the previous occurrence. As the essence of this structure relies on pointing back to previous occurrences, it is a Lempel-Ziv [23] bounded structure. In fact, if z is the number of phrases in the Lempel-Ziv parsing then the Block Graph uses $O(z \log^2 n)$ bits (where Lempel-Ziv requires $O(z \log n)$ bits) and carries out access queries in $O(\log n)$ time, however, they are not able to support more complex queries like rank and select.

A more recent and flexible data structure is Block Tree [6]. Block Trees are an adaptation of Block Graphs where each node represents a string and is divided into r non-overlapping equal-sized blocks.

[‡] `http://pizzachili.dcc.uchile.cl/repcorpus`

These point back to previous occurrences if there are any. This structure achieves $O\left(zr\log n\log_r \frac{n\log\sigma}{z\log n}\right)$ bits of space and access any symbol in time $O\left(\log_r \frac{n\log\sigma}{z\log n}\right)$. Moreover, they can extract a substring of size m in time $O\left(\log_r\left(\frac{n\log\sigma}{z\log n}\right)\left(\frac{m\log\sigma}{\log n}+1\right)\right)$. Experiments on repetitive collections show that Block Trees are an order of magnitude faster than a grammar-compressed representation [35] and use about the same space.

3 Problem Statement

We aim to research how to the use of Block Trees may lead to more efficient CSTs for repetitive collections. Concretely, we will replace each of the components of the CST of Sadakane [38] with Block-Tree-based structures:

- The tree topology represented with parentheses can be indexed with a Block Tree.
- The differentially encoded Suffix Array can be represented with a Block Tree.
- The differentially encoded LCP Array can be represented with a Block Tree.

For the parentheses, the Block Tree will have to be adapted to efficiently handle the standard queries [29, Chapter 8] required to implement a CST. These are navigational queries, for example, for a node: go to its parent, to some ancestor, child or sibling, compute its height, number of nodes or leaves on its subtree, or find the lowest common ancestor with another node.

A differential encoding of an array of numbers consists of another array that replicates the first value of the original array and, for the rest, stores the differences between consecutive elements. For both differentially encoded arrays, the Block Tree will have to be adapted to answer queries related to the original arrays (Suffix and LCP).

These components will be compared against alternatives currently used in the implementations of the CSTs for repetitive collections mentioned in Section 2. The alternatives are listed below in the same order as before.

- The Grammar-Compressed Topology (GCT) used by Ordoñez [31].
- The Run-Length CSA (RLCSA) used by both Abeliuk [1] and Ordoñez [31], and a Grammar-Compressed differentially encoded Suffix Array, idea implemented by González et. al [16] but never tried on repetitive collections.
- The adapted bitvector H [10] used by both Abeliuk [1] and Ordoñez [31]. And a Grammar-Compressed differentially encoded LCP Array used by Abeliuk [1].

These alternatives will be tested on repetitive text collections from the *PizzaChilli* platform and their space and time performance will be compared. After implementing the components we will combine the best options into a new CST, implementation that will be left public.

Literature Support

Block Trees are known to compress well highly repetitive sequences [6] [35] [29, Section 13.3.2]. There are good reasons to expect them to perform well on the three CST components for repetitive texts:

- For the topology, it is known that the Suffix Tree topology of a repetitive text contains large isomorphic structures, and consequently the topology represented by parentheses is repetitive [29, Section 13.2.4]. This was already exploited by the CST implementation of Navarro and Ordoñez [31] where the topology was successfully grammar-compressed.
- Repetitive texts induce areas in the Suffix Array of equal arithmetic progression, thus the corresponding differential Suffix Array is repetitive. Differential encoding of the Suffix Array has been developed too, using Re-Pair to grammar-compress the differential array [15] [16]. However, it was not tested on repetitive texts.
- At the end of their paper, Fischer et. al [10] suggest that repetitions induced in the differential Suffix Array show up in a differential encoding of the LCP array too. This idea was exploited in the Repetition-Aware CST of Abeliuk et. al [1] to grammar-compress it.

On the other hand, Block Trees were shown to be advantageous compared to grammars on repetitive sequences [35], thus replacing grammars by Block Trees on these components seems promising, at least if the sequences are sufficiently repetitive. Still, there are several open questions, as we detail next.

Research Questions

- Is it possible to adapt Block Trees to implement a fully-functional tree? Can the necessary operations be solved efficiently? What is the time performance for different queries on the tree and the space usage on repetitive collections compared to using grammars?
- Is it possible adapt Block Trees for a differential representation of the Suffix Array? What is the resulting time and space behavior on repetitive collections? How it compares against a grammar-compressed differential encoding? How it compares against the RLCSA?
- Is it possible to adapt Block Trees for a differential representation of the LCP Array? What is the resulting time and space behavior on repetitive collections? How it compares against a grammar-compressed differential encoding? How it compares against the adapted bitvector H?
- How is the resulting CST obtained compared, as a whole, in time and space against other repetition-aware CSTs?

4 Hypothesis

It is possible to adapt Block Trees to implement the three components of the CST, and in some of those the use of Block Trees offers better space/time trade-offs than current solutions.

5 Objectives

General Objective

Advance the state-of-the-art on repetition-aware CSTs by using Block Trees in the implementation of its components.

Specific Objectives

- Design, implement, test and compare a new tree representation, based on a Block Tree adaptation to support the necessary operations over the parentheses representation of the tree topology.
- Design, implement, test and compare a new CSA version, based on the Block Tree compression of the differential encoding of the Suffix Array.
- Design, implement, test and compare a new LCP Array compressed version, based on the Block Tree compression of the differential encoding of the LCP Array.
- Leave public a new CST implementation based on the results of the comparisons of the components.

6 Methodology

The following is a linear sequence of steps considered to develop this work. The steps are not necessarily non-overlapping.

1. Bibliographic revision and documentation of the components proposed as well as alternatives.
2. Simple implementation of Block Trees capable of performing the operations specified on its original publication [6].
3. Design of the adaptation of Block Trees proposed to work with the tree topology.
4. Implementation of the design made in point 3.
5. Test and comparison of implementation of point 4 and the GCT.
6. Design and implementation of Block Trees to encode differential information of the Suffix Array.
7. Test and comparison of implementation of point 6, the RLCSA, and the grammar-compressed differential Suffix Array.
8. Design and implementation of Block Trees to encode differential information of the LCP Array.
9. Test and comparison of implementation of point 8, the adapted bitvector H, and the grammar-compressed differential LCP Array.
10. Implementation of a new CST, based on the comparisons of the components.
11. Compare, as a whole, the new proposed CST, the CST of Abeliuk [1], and the CST of Ordoñez [31].

All the tests will be executed on different types of repetitive collections presented in the *PizzaChilli* platform. Time and space performance of the structures will be properly documented.

All developed software will be written in C++ based on its efficiency, its versatility and its support in bioinformatics [11].

7 Expected Results

We expect to obtain a new Compressed Suffix Tree for repetitive collections built on top of the recent data structure Block Tree, as well as to understand which components are best implemented with which alternative between previous work or Block Trees. Some components, like tree topologies, will have independent interest besides Suffix Trees. We will not only contribute with theoretical developments, but also practical implementations and experimental comparisons to aid practitioners in deciding which structure to use.

8 Contributions

- A new functional compressed tree based on Block Trees.
- A new CSA based on Block Trees.
- A new compressed representation of the Longest Common Prefix array based on Block Trees.
- Experimental time and space comparisons of the different components used for implementing repetition-aware CSTs.
- A new repetition-aware CST based on Block Trees, and a public implementation.
- Recommendations on which CST to use depending on the repetitive collection, for practitioners.

References

[1] Andrés Abeliuk, Rodrigo Cánovas & Gonzalo Navarro (2013): *Practical compressed suffix trees*. Algorithms 6(2), pp. 319–351.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz & Enno Ohlebusch (2004): *Replacing Suffix Trees with Enhanced Suffix Arrays*. J. of Discrete Algorithms 2(1), pp. 53–86.

[3] Alberto Apostolico (1985): *The myriad virtues of subword trees*. In: Combinatorial algorithms on words, Springer, pp. 85–96.

[4] Djamal Belazzougui & Fabio Cunial (2017): *Representing the Suffix Tree with the CDAWG*. In: 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland, pp. 7:1–7:13.

[5] 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.

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

[7] David R. Clark & J. Ian Munro (1996): *Efficient Suffix Trees on Secondary Storage*. In: *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 383–391.

[8] Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J Puglisi & Jouni Sirén (2015): *Relative Suffix Trees*. *CoRR*. Available at `http://arxiv.org/abs/1508.02550`. To appear in The Computer Journal.

[9] Paolo Ferragina & Roberto Grossi (1999): *The String B-tree: A New Data Structure for String Search in External Memory and Its Applications*. *J. ACM* 46(2), pp. 236–280.

[10] Johannes Fischer, Veli Mäkinen & Gonzalo Navarro (2009): *Faster entropy-bounded compressed suffix trees*. *Theoretical Computer Science* 410(51), pp. 5354–5364.

[11] Mathieu Fourment & Michael R Gillings (2008): *A comparison of common programming languages used in bioinformatics*. *BMC bioinformatics* 9(1), p. 82.

[12] Travis Gagie, Pawel Gawrychowski & Simon J Puglisi (2011): *Faster Approximate Pattern Matching in Compressed Repetitive Texts.* In: *ISAAC*, Springer, pp. 653–662.

[13] Travis Gagie, Christopher Hoobin & Simon J. Puglisi (2017): *Block Graphs in Practice*. *Mathematics in Computer Science* 11(2), pp. 191–196.

[14] Travis Gagie, Gonzalo Navarro & Nicola Prezza (2017): *Optimal-Time Text Indexing in BWT-runs Bounded Space*. *CoRR*. Available at `http://arxiv.org/abs/1705.10382`.

[15] Rodrigo González & Gonzalo Navarro (2007): *Compressed Text Indexes with Fast Locate*. In: *Proceedings of the 18th Annual Conference on Combinatorial Pattern Matching*, CPM'07, Springer-Verlag, Berlin, Heidelberg, pp. 216–227.

[16] Rodrigo González, Gonzalo Navarro & Héctor Ferrada (2015): *Locally Compressed Suffix Arrays*. *J. Exp. Algorithmics* 19, pp. 1.1:1.1–1.1:1.30.

[17] Dan Gusfield (1997): *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA.

[18] 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.

[19] Sebastian Kreft & Gonzalo Navarro (2013): *On Compressing and Indexing Repetitive Sequences*. *Theor. Comput. Sci.* 483, pp. 115–133.

[20] Stefan Kurtz (1999): *Reducing the Space Requirement of Suffix Trees*. *Softw. Pract. Exper.* 29(13), pp. 1149–1171.

[21] Shanika Kuruppu, Simon J. Puglisi & Justin Zobel (2010): *Relative Lempel-Ziv Compression of Genomes for Large-scale Storage and Retrieval*. In: *Proceedings of the 17th International Conference on String Processing and Information Retrieval*, SPIRE'10, Springer-Verlag, Berlin, Heidelberg, pp. 201–206.

[22] N. Jesper Larsson & Alistair Moffat (1999): *Offline Dictionary-Based Compression*. In: *Proceedings of the Conference on Data Compression*, DCC '99, IEEE Computer Society, Washington, DC, USA, pp. 296–.

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

[24] 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.

[25] Udi Manber & Gene Myers (1990): *Suffix Arrays: A New Method for On-line String Searches*. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 319–327.

[26] Edward M. McCreight (1976): *A Space-Economical Suffix Tree Construction Algorithm*. *J. ACM* 23(2), pp. 262–272.

[27] J. I. Munro & V. Raman (1997): *Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs*. In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, IEEE Computer Society, Washington, DC, USA, pp. 118–.

[28] Gonzalo Navarro (2012): *Indexing Highly Repetitive Collections.* In: *IWOCA*, Springer, pp. 274–279.

[29] Gonzalo Navarro (2016): *Compact Data Structures: A Practical Approach*. Cambridge University Press.

[30] Gonzalo Navarro & Veli Mäkinen (2007): *Compressed Full-text Indexes*. *ACM Comput. Surv.* 39(1).

[31] Gonzalo Navarro & Alberto Ordóñez Pereira (2016): *Faster compressed suffix trees for repetitive collections*. *Journal of Experimental Algorithmics (JEA)* 21(1), pp. 1–8.

[32] Gonzalo Navarro & Kunihiko Sadakane (2014): *Fully Functional Static and Dynamic Succinct Trees*. *ACM Trans. Algorithms* 10(3), pp. 16:1–16:39.

[33] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai & Masayuki Takeda (2016): *Fully Dynamic Data Structure for LCE Queries in Compressed Space*. In: *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, pp. 72:1–72:15.

[34] Enno Ohlebusch, Johannes Fischer & Simon Gog (2010): *Cst++*. In: *String processing and information retrieval*, Springer, pp. 322–333.

[35] Alberto Ordóñez (2016): *Statistical and repetition-based compressed data structures*. Ph.D. thesis, Universidade da Coruña.

[36] Luís Russo, Gonzalo Navarro & Arlindo Oliveira (2008): *Fully-compressed suffix trees*. *LATIN 2008: Theoretical Informatics*, pp. 362–373.

[37] Luís M. S. Russo, Gonzalo Navarro & Arlindo L. Oliveira (2011): *Fully Compressed Suffix Trees*. *ACM Trans. Algorithms* 7(4), pp. 53:1–53:34.

[38] Kunihiko Sadakane (2007): *Compressed suffix trees with full functionality*. *Theory of Computing Systems* 41(4), pp. 589–607.

[39] Sarah A Tishkoff & Kenneth K Kidd (2004): *Implications of biogeography of human populations for'race'and medicine*. *Nature genetics* 36, pp. S21–S27.

[40] Esko Ukkonen (1992): *Constructing Suffix Trees On-Line in Linear Time*. In: *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I*, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, pp. 484–492.

[41] Peter Weiner (1973): *Linear Pattern Matching Algorithms*. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, IEEE Computer Society, Washington, DC, USA, pp. 1–11.