Research Proposal
Contextual Pattern Search on Grammar-based Indexes for Highly Repetitive Text

Diego Ortego Prieto
Department of Computer Science
University of Chile
Santiago, Chile
diego.ortego@ing.uchile.cl

1 Introduction

Compression methods for text are arranged into two major classes, those for regular text and those for highly repetitive text, text where many substrings repeat often with little to no changes. For these last family of texts, specialized compressors achieve far superior compression rates[8] and some can be extended to allow random substring access and pattern searching, directly on the uncompressed data structure[11].

Compression methods for highly repetitive text are themselves divided into two large groups of diverse algorithms: Those that indeed can implement efficient random access of any substring of the text, which are generally simpler; and other more compression-efficient algorithms that are too complex to permit efficient random access. This research is specifically concerned with Grammar-Based Compression methods, which belong to the Random Accessible group.

Random Accessible methods generally build some sort of parse tree structure for representing the repetitions inside the text, grammars in particular build what's called a Grammar Tree[11]. This structure can be upgraded very easily for acquiring random access in time proportional to the height of the tree, which can be balanced during construction[7, 13].

Some of these methods, and grammars-based ones specifically, can be further upgraded into *Text Indexes*. These are structures that allow efficient pattern search without decompression. The algorithm for this query on grammars is generally more complex and requires building an additional data structure, but can enumerate results in amortized logarithmic time[3].

There are three kinds of grammar, Classic Grammars, Run-Length Grammars and Locally Consistent Grammars, where both the Classic and Locally Consistent kinds can become indexes[3, 4] and are subsets of Run-Length Grammars, which themselves currently only have a theoretical index proposed in 2020[2].

Local consistency is a property of the Grammar Tree that enhances pattern search efficiency by reducing the number of pattern splits that must be tried, at the cost of a generally larger tree[11].

These indexes work great for solving classic pattern search queries, but inside a highly repetitive text collection many occurrences of a pattern probably appear surrounded by the same repeated context.

An alternative pattern search solution could be, then, *Contextual Pattern Search*: The query that receives a grammar-compressed text T, a pattern P and an integer l, and returns the first occurrence of P for every unique context it appears in within T. The context is defined as the two substrings of length l to the left and right of every occurrence.

This query was first proposed by Navarro[10] along with an efficient but storage-impractical algorithm built on a classic index, the R-Index[6].
The issue that this research attempts to tackle is proposing and implementing the first efficient algorithm for *Contextual Pattern Search* on Grammar Indexes of the Classic and Locally Consistent kind. The solution is expected to be more efficient and practical that the one proposed on R-Indexes.
The research's objectives can be summarized into: Present two implemented efficient algorithms for computing the query, one for Classic and one for Locally Consistent Grammar-Based Indexes, that take as much advantage as possible of the specific properties of both kinds of grammar. Then attempt to prove a worst-case time complexity for these solutions leveraging those same properties.
Therefore the expected results are two implementations that are at least empirically faster than the trivial solution of finding all classic occurrences of the pattern in the index, looking up their contexts, sorting them and filtering the repetitions.
Finally, the contributions of this research are: All algorithms and properties found, with the implementations published under a free software license, and a publication in a conference that addresses repetitive text indexing, text compression, or experimental algorithms.

2 State of the Art

2.1 Compression Methods

Current compression methods use a wide variety of techniques for achieving the lowest sized files possible. Among them, there are those that specialize on regular text, where the degree of substring repetition within the text is small, and those that specialize on compressing highly repetitive strings like DNA sequences and historic internet file collections.
The most used methods also allow for random accessing any substring (queried as a starting character position and the length to be read) of the text without needing to decompress it. Even more interesting still, most of these methods also allow for querying all the positions where a pattern occurs inside the text without decompression. Methods with this last operation are specifically addressed as *text indexes*.
Techniques for highly repetitive string compression can achieve file sizes way smaller than regular compression methods[8] when applied to strings that satisfy a set of different measures of repetitiveness[11]. For example, a classical compression method that relies on the relative frequency of each symbol in the text (to then assign shorter codes to the most frequent symbols) cannot deal neatly with two identical strings that have been concatenated together. The compression method simply cannot figure out that the string repeats itself twice, which leaves the relative frequencies of the symbols unchanged, compressing it into twice the size of the compressed unconcatenated string.

2.2 Highly Repetitive String Compression

Because of this limitations, many compression methods for highly repetitive strings have been proposed and are used, some with their very own method-specific measures of repetitiveness. The mechanisms for compressing and reconstructing the original string vary a lot, and can be roughly divided into two sets:

2.2.1 Random-Accessible Methods

These mechanisms assign some sort of identifier to each substring that repeats itself more than once, and then find smart ways of grouping these identifiers together on a parse-tree-like way.

The reconstruction of the string is straightforward after this has been done, for each identifier there is a unique substring that must be found and replaced until the original string is rebuilt.
Because methods feature straightforward rules for decompression, they also allow efficient random access to any substring of the original uncompressed text, without needing to decompress it in its entirety.
The most representative and powerful example of a random-accessible method is the construction of a context-free grammar that represents the text, with another interesting example of this group being the Block Tree[9]. Both methods can be enhanced to allow for random-access and substring search[11].

2.2.2 Non-Random-Accessible Methods

These mechanisms compute the original string from a set of expansion rules more powerful than those in context-free grammars. They go beyond assigning a unique identifier to every repeated substring, and can make it so that the identifier's substring depends on the context where it is used, needing further computation. This makes it significantly harder to extract any specific zone of the text without decompressing all of it.
These methods, since they are more powerful, achieve higher compression rates on many common string families, but attempting random access or pattern search with them is very challenging and impractical when compared to simpler alternatives. Hence, they are widely used, but only for compression.
Because of this added complexity, efficient random-access of substrings using these methods simply does not exist yet, unless the original text is decompressed first or the method is simplified in some way.
The most representative method family in this group is Lempel-Ziv Compression[15], which can only be enhanced with random access at the heavy cost of simplifying the compression[5] (and larger file sizes).
One other representative example of this group is the Macro Scheme method, which constructs a context-full grammar-like representation of the text[14].

2.3 Features of Random-Accessible Methods

Among these methods, as previously mentioned, variants of the context-free grammar representation of a text are considered the most powerful in terms of their random access capabilities (although there are better methods for minimizing storage)[11], and so will be used from this point onward as examples for the necessary concepts.
There are many random-accessible methods, but they tend to have the following things in common:

- These methods rely on parsing the string into all substrings that are repeated more than once.
- When the string has been parsed, the methods assign substrings to nodes in some sort of parse-tree-like structure, in grammar methods this is called the Grammar Tree.
- Within this structure, only the first appearance of each substring needs to be stored, since the nodes in the tree of the same type can be transformed into that same substring later, saving space.
- It is formally said, then, that these compression methods parse the original string into **phrases** represented and grouped by specific nodes in the tree.

As shown in the Grammar Tree of Figure 1 all individual phrases can be recognized upon construction of the tree and the gray segments are not saved because they can be inferred from the black segments.
Since this is a grammar, symbols A, B and C in Figure 1 are what is called "Non-Terminals": They are parts of the tree that will eventually need to be replaced by their corresponding substrings.

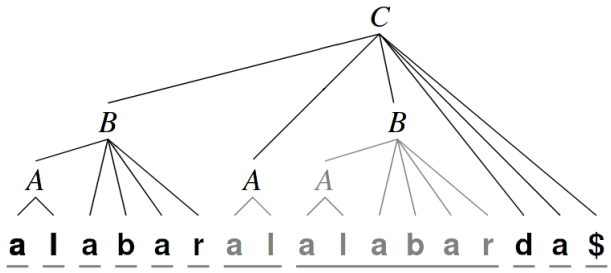

Figure 1: Edit of Figure 8 from Navarro's Survey[11], shows a Grammar Tree for the text *alabaralalabarda*, the symbols stored in memory are shown in black, while those that can be reconstructed while decompressing are shown in gray. Additionally, all individual phrases are underlined.

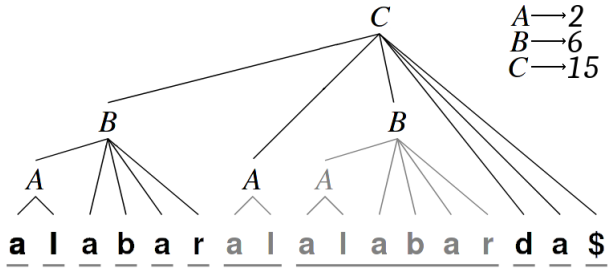

Figure 2: The Grammar Tree from Figure 1, including substring lengths for all kinds of non-terminal.

For the structure to become random-accessible it simply needs the modification shown in Figure 2. When storing the substring lengths of every kind of non-terminal, an algorithm can know in which node a certain position of the text is contained, and subsequently access any substring in time proportional to the height of the Grammar Tree.
The access time can become logarithmic on the size of the text when the Grammar Tree is balanced, which can be forced upon construction[13, 7], or even better than logarithmic, when using advanced techniques and complex additional structures[1].

2.4 Upgrading to Text Indexes

A text index is any structure that allows pattern searching within a representation of some text. Here the focus is on compressed indexes, but there are uncompressed examples too.
A classic and used example of an index is the R-Index presented by Gagie et al.[6]. This index is larger than what can be achieved with a Grammar-Based Index, but is also generally faster.
Grammar-Based Methods, and Parse-Based Methods in general, use an additional structure called a Parse Grid for allowing efficient substring search. As shown in Figure 3, the structure stores the indexes of every border where two phrases collide and keeps sorted lists of the reversed prefixes and suffixes of the text that end and start at these borders.
The algorithm for finding all occurrences of a particular pattern string in the text using this structure is rather simple, for every prefix of the string:

1. The prefix is reversed and binary-searched in the y axis of the grid, until the set of phrases that end with that prefix is found.

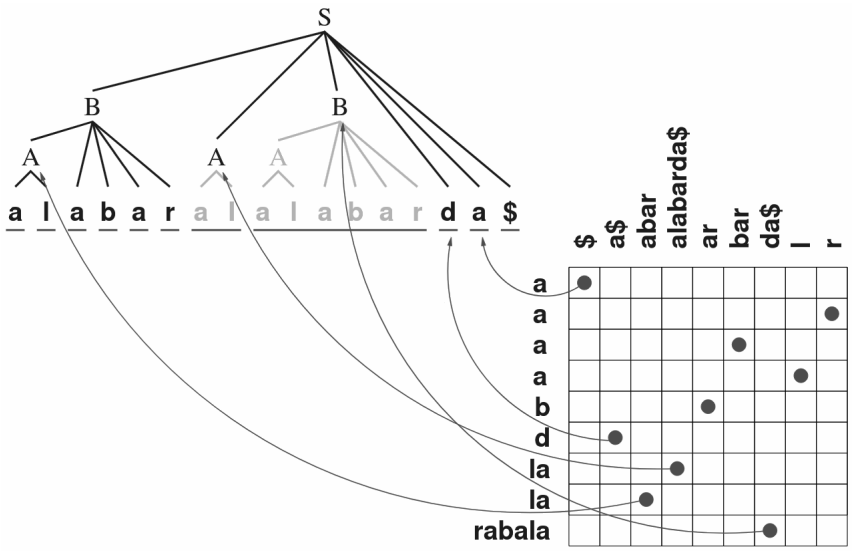

Figure 3: Edit of Figure in Slide 43 of Navarro's Talk[12], shows a Grammar Tree enhanced with a Parse Grid for the text *alabaralalabarda*. The x axis of the grid represents all alphabetically sorted suffixes that **start** at a phrase border. The y axis of the grid represents all alphabetically sorted, reversed, prefixes that **end** at a phrase border. The Grid itself stores pointers to the phrase borders where each prefix and suffix collide.

2. Then the rest of the string, the remaining suffix, is binary searched in the x axis of the grid, finding the phrases that start with that suffix.
3. The pointers in that rectangle of the grid point to all *primary occurrences* of the pattern string.
4. Then, the algorithm can climb the tree to search for all repeated appearances in the non-terminals that are not being explicitly saved in memory. This means that one must look for all other occurrences of a non-terminal that has a *primary occurrence*.

The example in Figure 4 shows the two primary occurrences of the pattern *ala* in text *alabaralalabarda* on positions 1 and 7 (starting from 1).
Looking at the text it is clear that there is also a third occurrence at position 9, but since the algorithm explicitly only searches for primary occurrences in the borders between phrases, and this third occurrence is *inside* a phrase, it must be found as a secondary occurrence.
Then Figure 5 shows the only secondary occurrence of *ala* in text *alabaralalabarda*, on position 9, found by looking for all repetitions of the non-terminal *B* (That contains the primary occurrence of position 1) in the tree.
Using this algorithm, Grammar Indexes can find all occurrences of a pattern in amortized logarithmic time[3].

2.5 Kinds of Grammar Index

As stated before, among Random-Accessible Methods, one of the most used and well-studied groups are those that construct a context-free grammar for representing the text. These are widely considered to be the most powerful method as well.

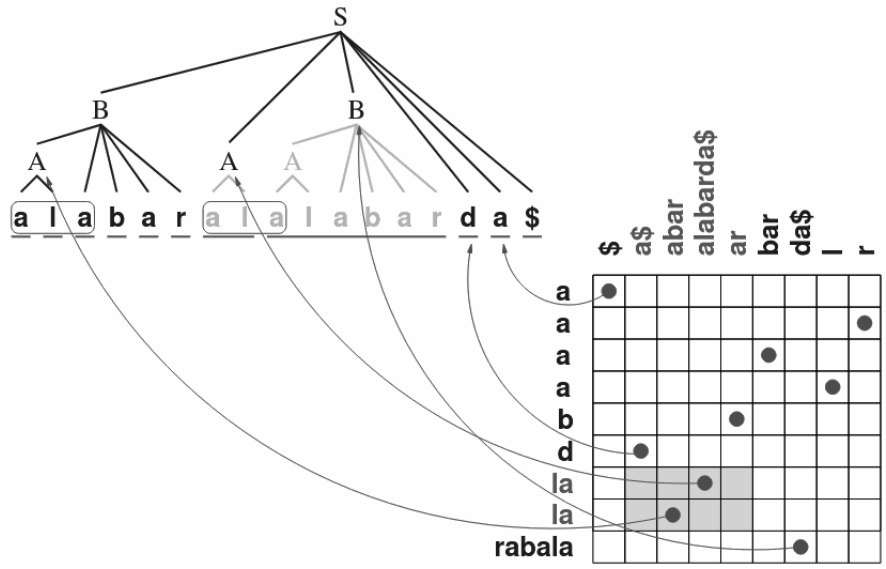

Figure 4: Edit of Figure in Slide 43 of Navarro's Talk[12], shows a Grammar Tree enhanced with a Parse Grid for the text *alabaralalabarda*, marking all primary occurrences of the pattern *ala* in the text.

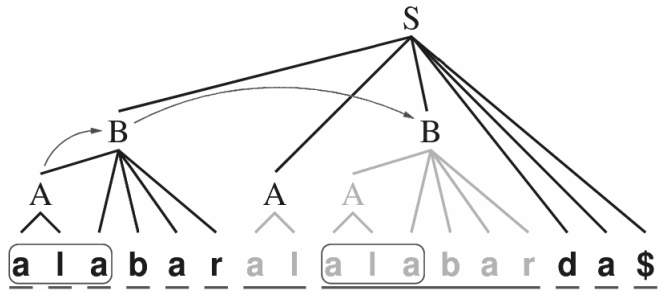

Figure 5: Edit of Figure in Slide 45 of Navarro's Talk[12], shows a Grammar Tree for the text *alabaralalabarda*, marking how to find the only secondary occurrence of the pattern *ala* in the text.

Grammars have such a high status because they are simultaneously conceptually simple to understand, very memory efficient, and easily upgradeable to indexes. They are, though, generally slower than some of the more memory-intense methods like R-Indexes.

There are three main kinds of Grammar Representations:

1. Classic Grammars: These correspond to the examples used above.

2. Run-length Grammars: A super-set of Classic Grammars, these are grammars that can represent a substring that is concatenated to itself many times as a non-terminal of the form $A^8 := AAAAAAAA$.

3. Locally Consistent Grammars: A subset of Run-Length Grammars, these are grammars that satisfy an extra property, the sub-trees of the Grammar Tree expanding to two identical substrings $S[i..j] = S[i'..j']$ are identical except for the $O(1)$ nodes on the left and right in each level of the sub-trees[11].

3 Problem Statement

When searching for a particular pattern in a highly repetitive text, all occurrences of this pattern are returned, even when they come from parts of the text that look generally the same.
That kind of search, though, is useful but limiting when the string is indeed very repetitive, as many of the search results will appear in similar context within the repeated substrings that comprise it[10]. For example, if a particular string is constructed by repeating a substring 10 times, all patterns that occur k times within the substring will have $10k$ occurrences in the whole text.
This is specially frustrating in cases like:

- DNA exon searches, where knowing the possible intron-exon-intron combinations in a particular sequence requires listing and sorting all the numerous occurrences of the exon pattern in the entire sequence.
- Searching for mentions of some sentence in the historic archive of some wiki, where many instances of the sentence are probably unchanged between different versions of the same overall page.

Therefore, as proposed by Navarro[10], a new type of query could be of use.

Given a text T, a pattern P and a length ρ

- Let O be the set of positions of all occurrences of pattern P in the original text T.
- Let $T' := \$^{\rho} T \$^{\rho}$, be T padded with ρ blank characters on each side.
- Using T' when looking at O_i, the i^{th} occurrence:
  - Let $L_i := T'[O_i]...T'[O_i + \rho - 1]$ be the substring of length ρ immediately to the left of the occurrence, in T' instead of T.
  - Let $R_i := T'[O_i + |P| + \rho]...T'[O_i + |P| + 2\rho - 1]$ be the substring of length ρ immediately to the right of the occurrence, in T' instead of T.
  - The pair (L_i, R_i) is known as the *context* of occurrence O_i.
- Then, we define C as the set of all occurrences of pattern P in text T that are the first in their respective *context*:

$$C := \{O_i \in O \mid \forall j \neq i \in [1, |O|],\ ((L_i, R_i) = (L_j, R_j) \wedge i < j) \vee (L_i, R_i) \neq (L_j, R_j)\}$$

  - Note that $C \subseteq O$.
  - An alternative definition is $C := \{O_i \in O \mid i = \min_{j \in [1,|O|]} \{j : L_i P R_i = L_j P R_j\}\}$

The *Contextual Pattern Search of pattern P and length ρ in text T* is then the query that enumerates the set C.
The proposed research is to extend Grammar-Based Indexes of both the Classic and Locally Consistent kind with an efficient *Contextual Pattern Search* query.

3.1 Literature Support

As stated before, this problem was formalized and proposed by Navarro. In his paper he offers a compelling case for the use of the query and an efficient implementation built on R-Indexes[10].

This solution is functional and efficient complexity-wise, but necessitates approximately double the storage of the original R-Index to work[10]. With the R-Index being already comparatively large on its own, it sets up an opportunity that Grammar-Based Indexes might offer a specially good solution for.
The Grammar-Based approach may be more practical, even if slower, than the existing R-Index implementation, since Grammar methods (specially of the Locally Consistent kind) already have an in-built concept of context:

- A non-terminal is contained by another when it is surrounded by a certain pattern of symbols, and that pattern of symbols is known by the parent non-terminal.
- Therefore, by getting closer and closer to the root of the Grammar Tree, a non-terminal will eventually be found to contain both the pattern and one of its contexts. An element of C has been found.
- Since the query only concerns itself with the first occurrence or the pattern in any specific context, it is unnecessary to search through the *secondary occurrences* of C. This reduces the computation time when compared to the classic pattern search.
- This process can then be repeated for all other contexts by following different paths in the tree.

Finding a way to efficiently traverse these non-terminals is one of the open problems we wish to address. Since in a Locally Consistent Grammar every context can only appear in a reduced amount of sub-trees in the Grammar Tree, we believe that these grammars might offer a specially efficient solution.

4 Research questions

- Can a Grammar-Based Index be implemented for solving *Contextual Pattern Search* queries with a better empirical time cost than the trivial solution of listing all occurrences, searching for their contexts, sorting the contexts and filtering repetitions?
- Can the performance of the query be improved by leveraging properties unique to Locally Consistent Grammars?
- Can it be proven that the proposed Grammar-Based Index has a better complexity cost than the trivial solution?

5 Hypothesis

A Grammar-Based Index can be implemented, such that it can execute *Contextual Pattern Search* queries more efficiently than the trivial solution of using a normal grammar-based index to list all occurrences of the pattern and then filter out the ones with a repeated context.

6 Objectives

6.1 General Objective

Propose and implement a grammar-based index that can do contextual pattern search better than the trivial solution of listing all occurrences and filtering repeated contexts.

6.2 Specific Objectives

1. Develop a basic contextual index on top of a Grammar Index.
2. Develop a basic contextual index on top of a Locally Consistent Grammar Index.
3. Find properties in both kinds of grammar indexes that could allow for faster contextual search algorithms.
4. Develop optimized contextual indexes that take advantage of these properties.
5. Evaluate whether these properties can lead to proving that the algorithm's complexity is be better than the trivial implementation's complexity.

7 Methodology

The steps to take are:

1. Become familiarized with Alejandro Pacheco's[3] implementation of a Grammar Index.
2. Implement a trivial contextual index on top of Pacheco's implementation, that simply finds the pattern and climbs the grammar tree looking for all non-terminal nodes that contain different contexts, and then reports the locations of these non-terminal nodes.
3. Benchmark this solution and the trivial solution, to quantify their difference in performance, by executing queries on a collection of Grammar-compressed DNA sequences or historic document databases.
4. Become familiarized with Diego Diaz's[4] implementation of a Locally Consistent Grammar Index.
5. Mirror the previous contextual index implementation on top of Diaz's index, which should lead to less repeated non-terminals with the same context and a more efficient result.
6. Benchmark this solution as well, running on the same previously mentioned dataset.
7. Study the search mechanism, looking for additional structures or other modifications that might enhance performance with or without an additional storage cost.
8. Propose concrete modifications to the trivial contextual indexes that could leverage whichever properties were found.
9. Implement the more efficient contextual indexes according to these modification proposals.
10. Benchmark the new solutions in the same manner as the previous ones.
11. Attempt to prove a worst-case complexity order for the improved algorithm, taking all modifications and found properties into account.
12. Redact the publication explaining the designed algorithms, related properties, implementation considerations, and results obtained along the way.

8 Expected results

At least two implementations of a grammar-based index that can execute Contextual Pattern Search queries better than the trivial solution of listing all occurrences and filtering repeated contexts:

- One implementation for Classic Grammar Indexes.
- One implementation for Locally Consistent Grammar Indexes.

9 Contributions

All algorithms, heuristics or helpful structures, as well as all implemented code will be published online in a freely accessible, readable, modifiable and usable manner.
Depending on the quality and complexity of the proposed algorithms, the resulting article will be submitted to one or more of the following academic conferences:

- SPIRE: String Processing and Information Retrieval Conference.
- DCC: Data Compression Conference.
- CPM: Combinatorial Pattern Matching Conference.
- SEA: Symposium on Experimental Algorithms Conference.
- ALENEX: Symposium on Algorithm Engineering and Experiments Conference.
- ESA B: European Symposium on Algorithms Conference, Engineering and Applications Track (Track B).

References

[1] Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj & Søren Vind (2017): *Fingerprints in compressed strings*. *Journal of Computer and System Sciences* 86, pp. 171–180, doi:10.1016/j.jcss.2017.01.002.

[2] Anders Christiansen, Mikko Ettienne, Tomasz Kociumaka, Gonzalo Navarro & Nicola Prezza (2020): *Optimal-Time Dictionary-Compressed Indexes*. *ACM Transactions on Algorithms* 17, pp. 1–39, doi:10.1145/3426473.

[3] Francisco Claude, Gonzalo Navarro & Alejandro Pacheco (2021): *Grammar-compressed indexes with logarithmic search time*. *Journal of Computer and System Sciences* 118, pp. 53–74, doi:https://doi.org/10.1016/j.jcss.2020.12.001. Available at `https://www.sciencedirect.com/science/article/pii/S0022000020301136`.

[4] D. Díaz-Domínguez, G. Navarro & A. Pacheco (2021): *An LMS-based Grammar Self-index with Local Consistency Properties*. In: *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 12944, pp. 100–113.

[5] Héctor Ferrada, Travis Gagie, Simon Gog & Simon J. Puglisi (2014): *Relative Lempel-Ziv with Constant-Time Random Access*. In Edleno Moura & Maxime Crochemore, editors: *String Processing and Information Retrieval*, Springer International Publishing, Cham, pp. 13–17.

[6] T. Gagie, G. Navarro & N. Prezza (2020): *Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space*. *Journal of the ACM* 67(1), p. article 2.

[7] Moses Ganardi, Danny Hucke, Artur Jez, Markus Lohrey & Eric Noeth (2015): *Constructing small tree grammars and small circuits for formulas*.

[8] Roberto Grossi, Ankur Gupta & Jeffrey Vitter (2002): *High-Order Entropy-Compressed Text Indexes*. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, doi:10.1145/644108.644250.

[9] G. Navarro (2017): *A Self-Index on Block Trees*. In: *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 10508, pp. 278–289.

[10] Gonzalo Navarro (2020): *Contextual Pattern Matching*. In Christina Boucher & Sharma V. Thankachan, editors: *String Processing and Information Retrieval*, Springer International Publishing, Cham, pp. 3–10.

[11] Gonzalo Navarro (2021): *Indexing Highly Repetitive String Collections, Part II: Compressed Indexes*. *ACM Comput. Surv.* 54(2), doi:10.1145/3432999. Available at `https://doi.org/10.1145/3432999`.

[12] Gonzalo Navarro (2021): *Talk: Repetitiveness and Indexability*. Available at `https://youtu.be/8twjSv39-c8`. Talk archived by Stanford Research Talks.

[13] Hiroshi Sakamoto (2005): *A fully linear-time approximation algorithm for grammar-based compression*. *Journal of Discrete Algorithms* 3(2), pp. 416–430, doi:https://doi.org/10.1016/j.jda.2004.08.016. Available at `https://www.sciencedirect.com/science/article/pii/S1570866704000632`. Combinatorial Pattern Matching (CPM) Special Issue.

[14] James A. Storer & Thomas G. Szymanski (1978): *The Macro Model for Data Compression (Extended Abstract)*. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, Association for Computing Machinery, New York, NY, USA, p. 30–39, doi:10.1145/800133.804329. Available at `https://doi.org/10.1145/800133.804329`.

[15] Jacob Ziv & Abraham Lempel (1977): *A universal algorithm for sequential data compression*. *IEEE Transactions on information theory* 23(3), pp. 337–343.