IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received June 11, 2020, accepted July 27, 2020, date of publication August 3, 2020, date of current version August 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3013676

A Fast Algorithm for the Largest Area First Parsing of Real Strings

IVAN KATANIĆ[1], STRAHIL RISTOV[2], AND MARTIN ROSENZWEIG[3]
[1]Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
[2]Ruer Bošković Institute, 10000 Zagreb, Croatia
[3]Department of Mathematics, Technische Universität München, 80333 Munich, Germany

Corresponding author: Strahil Ristov (ristov@irb.hr)

This work was supported in part by the Croatian Science Foundation under Grant IP-2018-01-7317, and in part by the European Regional Development Fund [DATACROSS] under Grant KK.01.1.1.01.0009.

ABSTRACT The largest area first parsing of a string often leads to the best results in grammar compression for a variety of input data. However, the fastest existing algorithm has $\Theta(N^2 \log N)$ time complexity, which makes it impractical for real-life applications. We present a new largest area first parsing method that has $O(N^3)$ complexity in the improbable worst case but works in the quasilinear time for most practical purposes. This result is based on the fact that in the real data, the sum of all depths of an LCP-interval tree, over all of the positions in a suffix array of an input string, is only larger than the size of the input by a small factor α. We present the analysis of the algorithm in terms of α, and the experimental results confirm that our method is practical even for genome sized inputs. We provide the C++11 code for the implementation of our method. Additionally, we show that by a combination of the previous and new algorithms, the worst-case complexity of the largest area first parsing is improved by a factor of $\sqrt[3]{N}$.

INDEX TERMS Greedy grammar compression, largest area first parsing, dynamic text indexing, enhanced suffix array.

I. INTRODUCTION

Finding the repetitions in a string is among the most researched tasks in stringology. One important usage is the representation of a string with the smallest context-free grammar (CFG), either for compression purposes or for obtaining insight into the string's structure [1]–[3], or for a construction of grammar-based compressed indices [4]. Grammar text compression is a compression procedure where repeated substrings in a string are replaced with the production rules (non-terminals), and the string is represented with a CFG that has the exact input string as the only product. To achieve a compression, the replaced substrings must have the length of at least two symbols and occur at least twice in the string. Finding the smallest CFG is a problem known to be NP hard [5]–[7]. As a result, various heuristics are used in order to find the set of repeated substrings that would yield the best approximation of the smallest grammar. In practice, there are three prominent global strategies for the parsing of a string: *most frequent first* (MFF), *longest first* (LF), and *largest area first* (LAF) [2]. Additionally, a grammar can be obtained from the output of some members of the LZ family of online algorithms, which can be described as the *longest previous* sequential parsing, and has been used in theoretical analysis of the approximation bounds [6], [8]. In addition to parsing, the actual compression requires the additional coding of the output, which is beyond the scope of this article. A comprehensive overview of parsing methods for grammar text compression can be found in [6], [9]. Here, we shall summarize the time complexity results for the three listed variants.

MFF parsing replaces substrings with rules in order of their frequency. A recursive algorithm for this task that starts with the most frequent pairs of characters and runs in linear time was introduced in [10]. The LF method finds the current longest repeated substrings at any point in the algorithm execution and iteratively replaces them with rules [11]. A linear time worst-case algorithm that uses a suffix tree and allows for the creation of shorter rules within the already existing rules was first described in [12]. The LAF method iteratively replaces the substrings that provide the highest compression gain at any point in the execution [13]. The existence of a worst-case linear time algorithm for this task is an open problem. The best existing method, based on [14], has $\Theta(N^2 \log N)$ time complexity.

In this paper, we present an algorithm that performs largest area first parsing in quasilinear time on a variety of real-life data. The algorithm has $O(N^3)$ worst-case time complexity

The associate editor coordinating the review of this manuscript and approving it for publication was Donghyun Kim.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

in the case of an input that consists of a repetition of only one symbol, but it exhibits approximately linear behavior in experiments on a wide range of standard test files. We propose that, under some realistic assumptions, the average time complexity of our method is $O(N \log^2 N)$.

In addition, we show that by combining our algorithm with that from [13] into a hybrid method, we can obtain a better worst-case complexity for LAF parsing.

In the Section II, we give a short outline of the largest area first method as well as the problem of the dynamic indexing of a text. In Section III, we describe our algorithm, and in Section IV, we present the time complexity analysis together with the experimental results. The proof of the new worst-case result is presented in Section V. We conclude in Section VI.

II. LARGEST AREA FIRST GRAMMAR TEXT COMPRESSION AND DYNAMIC TEXT INDEXING

The LAF parsing method was introduced in [13], where it was referred to as the method of *steepest descent*. Alternative names used in the literature to describe the same concept include *greedy* and *compressive*. The area that is covered by a specific repeated substring is defined as the product of the number of nonoverlapping occurrences of the substring and its length. Replacing all instances of a substring that covers the largest area with a rule gives the largest compression gain at each iteration. Due to the cost of storing a rule, in the cases when the same area is covered by different substrings, a better compression is achieved with longer substrings and fewer instances. The method is reported in [13] and [2], and more recently in [15], as the best among grammar compression methods, in terms of the compression efficiency, for the most types of input data. A recent practical result is reported in [16], where LAF parsing has been used as the first step in a competitive general purpose compressor.

As with the other parsing methods, the main problem regarding the time complexity of the procedure is how to efficiently find the next set of repeated substrings at the top of the priority queue according to the given criteria. In this case, after the replacement of all the occurrences of one substring with a rule, we need to recalculate the areas of all remaining repetitions that were affected in the current iteration and find the new largest area.

This is an instance of the problem of dynamic text indexing. The standard structures used for static text indexing are the *suffix tree* (ST) and *suffix array* (SA) [17] (all string algorithm data structures mentioned in the article will be defined in subsection III.B). Both ST and SA can be constructed in linear time and used to efficiently find all occurrences of a substring in a string. However, in dynamic settings, when the text changes, the index must be updated, and the best method for the updates depends on the application. With LAF parsing, there are two problems. First, the sizes of areas are calculated only from the number of nonoverlapping occurrences, and second, the index must be updated after each substitution.

In the original paper [13], a structure called the *minimal augmented suffix tree* (MAST) is used to find the set of repeated nonoverlapping substrings that covers the largest area of the text in time linear with the number of substring occurrences. This augmented suffix tree can be constructed in $O(N \log N)$ time following [14]. However, after each substitution of an area, updating the index means rebuilding the suffix tree from scratch. Therefore, the expected time complexity for this approach cannot be less than $O(N^2 \log N)$. A different approach is possible using the method for nonoverlapping indexing described in [18] by employing a slightly modified suffix tree, which can be constructed in $O(N)$ time. However, this approach does not include the time necessary to find the substring that covers the largest area. As a result, no known method exists based on the reconstruction of a suffix tree that is faster than $\Theta(N^2 \log N)$.

For some applications, an ST can be updated in amortized linear time when the updates are localized or otherwise restricted as in [12]. Unfortunately, this does not seem to be the case with LAF parsing since a method for an efficient (amortized constant time) global suffix tree update (of all occurrences of a substring) is not known. Standard suffix arrays are even harder to update because they require the reordering of a contiguous array [19], [20].

We propose that for a subset of dynamic problems, the difficulty of dynamic index updates can be overcome by using a static suffix array in conjunction with auxiliary structures that may support updates in constant time. This approach is possible when each substitution of a substring with a rule affects only a limited area of the string (and its index).

A suffix array, enhanced with the additional data structures, can support every type of search over a string within the same time complexity as that of a suffix tree [21]. Since an SA is simpler and requires considerably less space than an ST, the SA has become the structure of choice in string processing applications. Nevertheless, the ST still has an advantage in some dynamic applications because less work is required to update the links of a branch of a tree than to reorder an array. Favoring the SA approach, we have found that in some dynamic applications, it is possible to use a suffix array without the need for changes in the array itself, and instead, the updates may be performed in the fast auxiliary structures. We have established that this approach is possible when an application does not require fully random updates but when the changes follow a predefined order instead. We focus on the cases where the number of algorithm steps can be determined using the sum of depths in *LCP-interval tree* [21] over all of the positions in the suffix array. We have previously explored this property to develop algorithms that run in quasilinear time for two of the other dynamic string parsing problems, the longest first grammar compression using SA [22] and the *longest previous with updates* used in a variant of LZ parsing [23]. In the present work we show how, for the most practical purposes, LAF parsing can be performed in the

number of steps that is bounded within a polylog by the input size.

III. THE ENHANCED SUFFIX ARRAY ALGORITHM FOR LAF PARSING

A. OVERVIEW OF THE NEW ALGORITHM

The high-level organization of our method is presented with the following steps:

1) Find the initial areas that can be replaced with a rule and sort them according to their size into a list.
2) Replace all instances of a substring that cover the largest area with a rule.
3) At each replacement of an instance of a substring with a rule, modify the size of all areas that are affected and move the affected areas to maintain the sorted order in the list.
4) Remove the largest area from the list.
5) Repeat starting from step 2 until there are no more areas in the list.

The main difference between our algorithm and the original algorithm from [13] lies in step 3. Instead of rebuilding the whole index after the replacement of the entirety of the largest area with rules, we perform the local updates after the replacement of each single substring belonging to the currently largest area. These updates consist of the following:

- Find all areas that are affected with the replacement of one substring with a rule.
- For each affected area, calculate the reduction in the area size and update the area position in the sorted list accordingly.

The key to efficiently performing local updates is in efficiently finding all affected areas. To do so, we use the suffix array enhanced with additional data structures, where the areas covered by the same substring correspond to the intervals in the LCP-interval tree. In addition, we use the suffix array, in conjunction with some additional data structures, to find the area sizes without overlaps. Cumulatively, this approach leads to the low amortized complexity of our method.

B. INTRODUCTORY REMARKS AND NOTATION

We define the standard string algorithm structures used in our method. Let T be an input string of the length $|T|$ over an alphabet A of the size $|A|$. A common procedure with string algorithms is to append to T a character from A that does not appear elsewhere in T and is lexicographically the smallest (or the largest) of all the characters in T. If we denote this character with #, then a suffix array $SA(T\#)$ is an ordered table of integers from 0 to $|T|$ that correspond to the lexicographical ordering of the suffixes of $T\#$. The *inverse suffix array* SA^{-1} is a table of integers such that $SA^{-1}[SA[i]] = i$, for $0 \leq i \leq |T|$. The *longest common prefixes* (LCPs) of the suffixes at the consecutive positions in $SA(T\#)$ are stored in the *LCP array*. $LCP[0] = 0$, and $LCP[i]$ is the length of the shared prefix of the suffixes of T starting at positions $SA[i]$ and $SA[i-1]$. A set of successive positions in SA that has an LCP value equal to, or larger than, the given threshold minimal value constitutes an *LCP-interval*. Intervals that have a higher LCP value can be nested in intervals that have lower values. The hierarchy of the LCP-intervals is presented with the *LCP-interval tree* [21]. In the rest of the text, we use the term *interval* to denote an LCP-interval in a suffix array.

Let $I_i.lcp$ denote the LCP length of interval I_i. We say that a substring s_i is a member of an LCP-interval I_i if it has the length of $I_i.lcp$ and starts at the position p in T such that $a = SA^{-1}[p], a \in I_i$. We also say that such positions p and substring members starting at p belong to I_i.

Let $I_i.weight$ denote the weight of I_i. The weight of an interval is calculated as the sum of the lengths of all nonoverlapping members of the interval. In our algorithm, we reduce the lengths by 1, i.e., we sum up $(I_i.lcp - 1)$, in order to account for the cost of storing the symbols for the rules that replace interval members.

Let S denote the sum of depths of the LCP-interval tree for all positions in SA, excluding the nodes that have LCP values smaller than 2. Let d_i denote the depth of the LCP-interval tree at the position i in SA. Then, S is defined as:

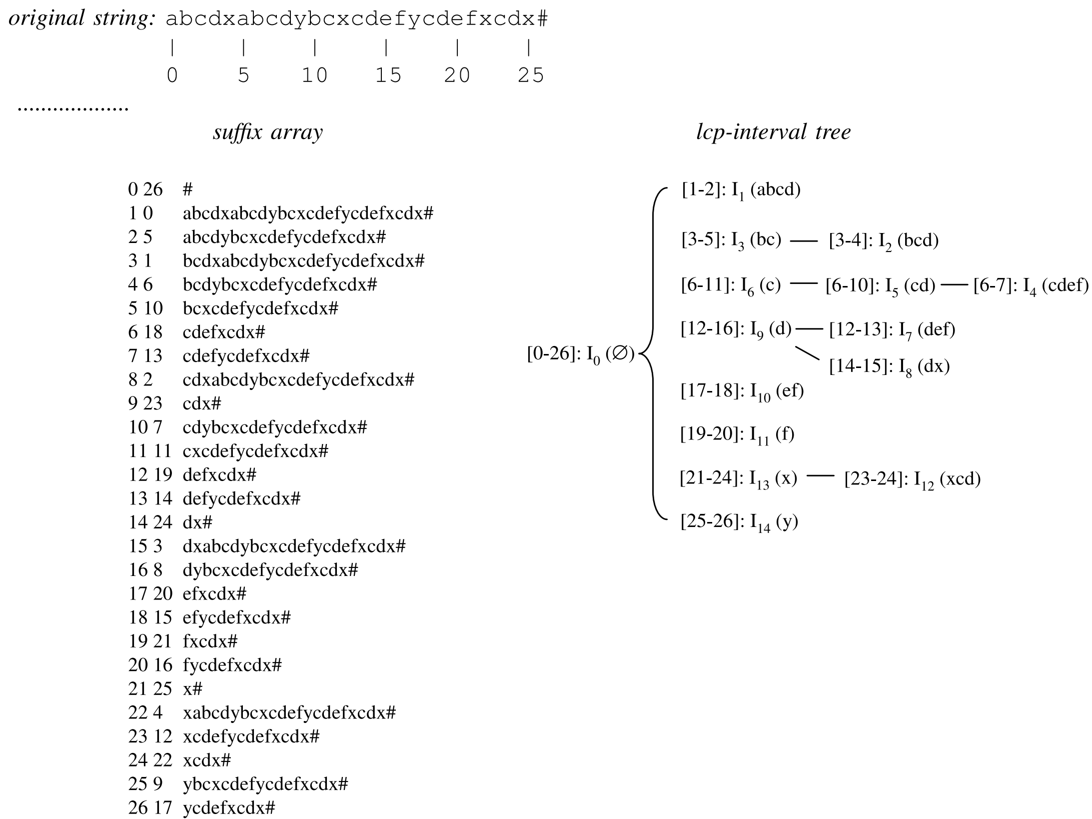
$$S = \sum_{i=1}^{|T|}(d_i - c_i)$$

$$c_i = \begin{cases} 1 & \text{if there exists a node with LCP value = 1 at position } i \\ 0 & \text{otherwise} \end{cases}$$

The factor c_i compensates for the intervals with an LCP value of 1 since the repeated substrings are meaningful only if they are at least two symbols long. The value of S is essential to our approach because we have established that some problems requiring dynamic string indexing can be solved in the number of steps that is linear with S. The size of S depends on the characteristics of the source of T. In the worst case, when $|A| = 1$, S is $O(|T|^2)$. However, in most practical cases, S is conveniently small. If we denote with α the factor by which S is larger than $|T|$, then $S = \alpha|T|$ and $\alpha \ll |T|$ for all of the tested real-life data sets. α is equivalent to the average depth of the LCP-interval tree, corrected by the factor c_i. Furthermore, the structure of the LCP-interval tree is equivalent to the structure of a suffix tree for a given T. Therefore, $\alpha_T \approx$ (average depth of $ST(T)$) - 1.

C. EXAMPLE GRAMMAR PRODUCTION

Fig. 1 shows the suffix array, the corresponding LCP-interval tree, the initial parsing of the areas, the final grammar and the compressed representation of an example input string. To facilitate the explanation of the algorithm, we establish the equivalence between the rules and the intervals in the LCP-interval tree. The areas covered by the repeated substrings in the string correspond to the LCP-intervals in the SA. The area sizes are represented by interval weights. For the areas with the same weight, the order of priority is arbitrary. Such is the case with R_1 and R_2 in Fig. 1. We recall that we calculate weights by summing up the length of



....

initial parsing of potential rules, sorted by area size:

$R_1 = abcd\,|$ interval I_1, $I_1.weight$ = 6, positions : 0, 5
$R_2 = cdef\,|$ interval I_4, $I_4.weight$ = 6, positions : 13, 18
$R_3 = cd\,|$ interval I_5, $I_5.weight$ = 5, positions : 2, 7, 13, 18, 23
$R_4 = bcd\,|$ interval I_2, $I_2.weight$ = 4, positions : 1,6
$R_5 = def\,|$ interval I_7, $I_7.weight$ = 4, positions : 14,19
$R_6 = xcd\,|$ interval I_{12}, $I_{12}.weight$ = 4, positions : 12, 22
$R_7 = bc\,|$ interval I_3, $I_3.weight$ = 3, positions : 1, 6, 10
$R_8 = dx\,|$ interval I_8, $I_8.weight$ = 2, positions : 3, 24
$R_9 = ef\,|$ interval I_{10}, $I_{10}.weight$ = 2, positions : 15, 20

....

final grammar:

$R_1 \to abR_3$
$R_2 \to R_3ef$
$R_3 \to cd$
b, c, x, y

....

compressed string: R_1xR_1ybcxR_2yR_2xR_3#

FIGURE 1. The example of the employed data structures and the output of the largest area first parsing of a string. The areas covered by the same substring correspond to the intervals in the LCP-interval tree. The final grammar consists of three rules R_{1-3} and four terminals *b, c, x* and *y*. For the formation of a shorter rule that is a part of the already formed longer rules, the existing rules are modified to include the new rule. Intermediate steps between the initial parsing and the final grammar are shown in Fig. 2.

repeated substring - 1. Therefore, although rule R_3 covers a larger area than rules R_1 and R_2, it achieves less compression when we factor in the cost of storing the rule symbol.[1]

The grammar consists of rules that correspond to LCP-intervals listed in order of their weight and the explicit set of terminals that are not a part of any rule. In each iteration, a new rule overwrites the previous shorter rule if the previous rule is entirely included in the new longer rule. The intervals are annotated with their corresponding weights as well as with the explicit substring that corresponds to the interval, together with the list of the positions in the string where the substring is replaced with the rule.

[1]The calculation of the achieved compression should take into account the cost of storing the definition of the rule, too, therefore reducing the summed up weight by $I_i.lcp$. In that case, R_3 would have priority over R_1 and R_2 in Fig. 1. However, we choose to ignore the cost of storing the definition of a rule since we do not know their final sizes in advance.

A position in T may belong to multiple LCP-intervals with different lengths. For example, the position 6 in T that corresponds to position 4 in *SA* belongs to the intervals I_3 (*bc*) and I_4 (*bcd*).

As an example for the calculation of S, the depth of the LCP-interval tree at position 7 in the suffix array is 3, but this position contributes with 2 to S since the tree includes the interval I_6 that has an LCP value of 1. Summing up over all positions in Fig. 1 gives $S = 22$. The sum of the depths of the LCP-interval tree is equal to the sum of the number of positions that belong to an interval with $LCP > 1$ over all LCP-intervals; i.e., S equals the number of all repeated substrings in a string.

The three intermediate steps between the initial parsing and the final grammar are presented in Fig. 2. Fig. 2 shows the state of the sorted list of intervals after the replacement of the whole of an interval with a rule. The first step shows the state after the replacement of substrings *abcd* with rule R_1, the second step shows the state after the replacement of substrings *cdef* with rule R_2, and the third step shows the state after the replacement of substrings *cd* with rule R_3. The substrings are replaced with a rule both in the input string and in the definitions of the previous rules. Only the repeated substrings can form a rule, and therefore, the weight is calculated only if more than one instance of a substring exists, counting both the instances of a substring in the text and in the definitions of previous rules. Replacing a member of an interval with a rule deletes members of other intervals. If an interval is reduced to only one member, or none, the weight is set to zero, which effectively removes this interval from the list. In the first step in Fig. 2, intervals I_2, I_3 and I_8 are reduced to one or zero members and consequently removed from the list. In the second step, this happens with I_7, I_{12} and I_{10}.

1) OVERLAPS

Consider the string *abababa$*. Although there are three repetitions of substring *aba*, only two can be replaced with a rule. In our algorithm, we denote the instances of the repeated substrings, from left to right, as *active* if they do not overlap with the previous active instance of the same substring. Only the active members of the intervals are summed up in the calculation of the interval's weight; i.e., only the first and the third *aba* in *abababa* contribute to the area covered by *aba*.

D. INITIALIZATION PHASE OF THE ALGORITHM

The first task in our method is to calculate interval weights and sort the intervals in order of diminishing weights. To do so, we need to set up necessary data structures, the most important of which we list below.

- *SA and LCP array* - Temporary data structures. A suffix array is constructed as the first step for producing the inverse SA and LCP array and then discarded. An LCP array is used for the construction of the LCP interval tree and then discarded. We use the *sais* [24] method for SA construction and the Kasai [25] method for LCP array construction, both in linear time.

Rule/Interval	substring	Interval weight	positions
R_2/I_4	cdef	6	13,18
R_3/I_5	cd	4	13,18,23,R_1
R_5/I_7	def	4	14,19
R_6/I_{12}	xcd	4	12,22
R_9/I_{10}	ef	2	15,20
R_4/I_2	bcd	0	$\emptyset$
R_7/I_3	bc	0	10
R_8/I_8	dx	0	24
....			
formed rules: R_1=abcd			
R_3/I_5	cd	3	23,R_1,R_2
R_5/I_7	def	0	$\emptyset$
R_6/I_{12}	xcd	0	22
R_9/I_{10}	ef	0	$\emptyset$
....			
formed rules: R_1=abcd, R_2=cdef			
No more valid intervals.			
....			
formed rules: R_1=abR_3, R_2=R_3ef, R_3=cd			

FIGURE 2. The example grammar production in steps.

- *Inverse SA* - Constructed in linear time from *SA* and used in LCP array construction. At each position p in T, we use the inverse SA and LCP-interval tree to find which LCP-intervals $SA^{-1}[p]$ belongs to.
- *LCP-interval tree* - The LCP-interval tree is implemented as two arrays: *TerminalNode*, which stores the index of the deepest interval that a position in *SA* belongs to, and *Parent*, which stores the interval's ancestor in the tree. The number of nodes in the LCP-interval tree is bounded by $O(|T|)$. However, the number of steps in the construction of the LCP-interval tree is $O(S)$ since that involves iteration over all nodes at each position in the LCP array.
- *IntervalMember* - Interval member data are stored in a two-dimensional array of structures *IntervalMember*, where for each interval member, we store its position in T and the *active/inactive* status. The number of interval members (repeated substrings in a string) equals S.
- *InversePositions* - A two-dimensional array that for each p stores the ranks of interval members starting at p for each interval that p belongs to. It is used to find the positions of two consecutive members of an interval. This approach is necessary in the weight update phase when we need to verify whether the deactivation of one interval member activates another member. A rank is stored for every member of each interval, and therefore, the size of *InversePositions* is S.
- *MemberIndex* - An array that stores the current member of an interval during the initialization process. It is discarded after the initialization.

Algorithm 1 Initialization phase. *overlap*(I, i) returns true if a member of I at the position i overlaps with the last previous active member of I.

```
   Input: T, InverseSA(T), LCP interval tree:
          TerminalNode and Parent arrays
 1 int IntervalMember[interval][MemberIndex].position ;
 2 bitflag IntervalMember[interval][MemberIndex].
   active ;
 3 int InversePositions[position][offset] ;
 4 int MemberIndex[interval]; (initialize to zeros)
 5 int Weight[interval]; (initialize to zeros)
 6 for i = 0 to |T|-1 do
 7     offset = 0 ;
 8     I = TerminalNode[InverseSA[i]] ;
 9     while I do
10         IntervalMember[I][MemberIndex[I]].
           position = i ;
11         if not overlap(I, i) then
12             Weight[I] + = I.lcp - 1;
13             IntervalMember[I][MemberIndex[I]].
               active = 1 ;
14         else
15             IntervalMember[I][MemberIndex[I]].active
               = 0 ;
16         InversePositions[i][offset] = MemberIndex[I] ;
17         MemberIndex[I] ++; offset ++ ;
18         I = Parent[I] ;
19 using pigeonhole sort on Weight table create
   IntervalsByWeights array;
```

- *IntervalsByWeights* - An array that stores intervals sorted by their weights. We implement this as a two dimensional array, where the rows are indexed by interval weight and the columns by the position of an interval in the row. Both of these indices are updated when the weight of an interval changes. The positions in a row for each interval are stored in a separate table. This implementation allows changes in the interval's location in the *IntervalsByWeights* in constant time.

Some more obvious auxiliary data structures are omitted for clarity. The essential pseudocode for the preprocessing phase is presented in Algorithm 1. The weight of each interval can be calculated in one pass over all of the substrings belonging to the interval if we access them in order of their positions in the string. In this way, it is possible to include only the nonoverlapping substring instances in the weights. Each substring in every interval that participates in the interval weight is marked as *active*. This information will be used later to determine the weights that need updates. To test for overlaps during the initialization, we need to keep an account of only the previous active position for each interval.

After the initial weights are calculated, the final step of the initialization is to sort the intervals according to their weights. This is done using the pigeonhole sort on the weights to produce the new *IntervalsByWeights* (IBW) array. The initial parsing of potential rules, sorted by area size shown in Fig. 1, is an example of the initial state of an IBW array.

The maximum weight an interval can have is $O(|T|)$; therefore, the sorting is performed in $O(|T|+|I|)$ steps, where $|I|$ is the number of LCP-intervals. $|I|$ is equal to the number of internal nodes in a suffix tree, which is bounded by $|T|$. Consequently, the overall number of steps for the production of the array of intervals sorted according to their weights excluding overlaps is $O(S + |T|)$.

It should be noted that although the LCP-interval tree does not include explicit nodes for all repetitions in a string as does the MAST structure used in the original paper [13], it can be shown that at each iteration, the area covered is of the same size when using both structures. The crux of the proof is the following: while with a string such as *ababa$* there wouldn't exist a node for the substring *ab*, a node would certainly exist that would lead to the same size area coverage, in this case for the substring *ba*. Since the LAF paradigm does not rank the areas of the same size, our algorithm is correct. More formally,

Lemma 1: The largest areas found in each iteration using the LCP-interval tree are of the same size as those found using MAST.

Proof:

Let $T[i, i+1, \ldots, i+j-1]$ denote the substring of T of length j starting at the position i in T. Then, let $T[p_1 \ldots p_1 + l - 1], \ldots, T[p_k \ldots p_k + l - 1]$ denote a set of k nonoverlapping substrings of T with length l starting at the positions $p_1, p_2 \ldots p_k$ that covers the currently largest area with the size of kl. Let c be the largest nonnegative integer such that $T[p_1+l \ldots p_1+l+c-1] = \ldots = T[p_k+l \ldots p_k + l + c - 1]$. Then, the substring $T[p_1 + c \ldots p_1 + c + l - 1]$ is present in the LCP-interval tree, and therefore, in our algorithm, we shall consider the set of nonoverlapping substrings $T[p_1 + c..p_1 + c + l - 1], \ldots, T[p_k + c..p_k + c + l - 1]$ with the same area size of kl.

□

E. MAIN SUBSTRING REPLACEMENT PROCEDURE

Let $s_{i,j}$ denote a substring of T that is a member of I_i and starts at the position j in T. Then, $s_{top,j}$ denotes a member of interval I_{top} that is currently at the top of the IBW. The main algorithm procedure consists of the following: 1) replacing each active instance j of a substring $s_{top,j}$ with a rule and 2) at each replacement, updating $I_A.weight$ and IBW for each interval I_A that is affected by the replacement. The potentially affected intervals are the following:

- All intervals I_A that have members starting at positions in T that are included in every active instance of $s_{top,j}$. These are the positions replaced with a rule.
- All intervals I_A that have members starting at the positions p_l to the left of j such that $p_l + I_A.lcp \geq j$. We call such positions the *affected left neighborhood*. The affected left neighborhood extends up to the first

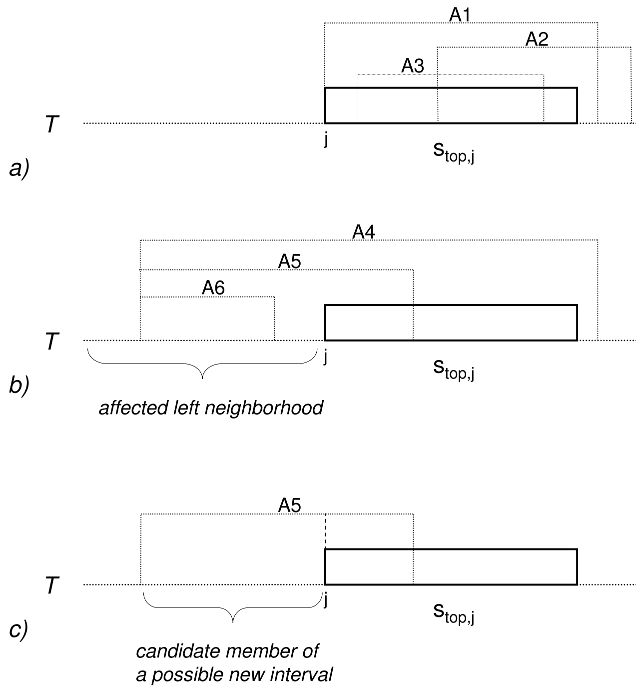


FIGURE 3. Positions and lengths of the members of the possibly affected intervals after the replacement of $s_{top,j}$ with a rule.

position that does not belong to any interval whose members overlap with $s_{top,j}$.

We list six different cases of intervals in the affected area regarding the position and the length of a string that belongs to the potentially affected interval relative to the replaced substring $s_{top,j}$. The cases of the affected intervals that occur within the replaced substring (A1-A3) are illustrated in Fig. 3a), and the cases that occur in the affected left neighborhood (A4-A6) are illustrated in Fig. 3b). With $startpos_A$ and $endpos_A$, we denote the starting and the ending positions of the substring that is a member of the affected interval, respectively.

- A1: $startpos_{A1} = j$; $endpos_{A1} \geq j + I_{top}.lcp$;
  affected interval weight change: $-(I_{top}.lcp - 1)$
- A2: $j < startpos_{A2} \leq (j + I_{top}.lcp - 1)$; $endpos_{A2} \geq j + I_{top}.lcp$;
  affected interval weight change: $-(I_{A2}.lcp - 1)$
- A3: $j \leq startpos_{A3} < (j + I_{top}.lcp - 1)$; $startpos_{A3} < endpos_{A3} \leq j + I_{top}.lcp - 1$;
  affected interval weight change: $-(I_{A3}.lcp - 1)$
- A4: $startpos_{A4} < j$; $endpos_{A4} \geq j + I_{top}.lcp - 1$;
  affected interval weight change: $-(I_{top}.lcp - 1)$
- A5: $startpos_{A5} < j$; $j \leq endpos_{A5} < j + I_{top}.lcp - 1$;
  affected interval weight change: $-(I_{A5}.lcp - 1)$
- A6: $startpos_{A6} < j$; $startpos_{A6} < endpos_{A6} < j$;
  affected interval weight change: *none*

Updating the weight of the affected interval I_A consists of reducing $I_A.weight$ by $I_A.lcp - 1$ or $I_{top}.lcp - 1$, but only if the substring at the processed position is active in I_A. If it is not, then $I_A.weight$ remains unchanged.

The active interval members in the affected area are deactivated in the cases A2, A3, and A5, and $I_A.weight$ is reduced by $I_A.lcp - 1$. Each time that a member of an affected interval I_A is deactivated, we must check if this activates some of the following substrings belonging to I_A that were inactive because of the overlap. If that is the case, $I_A.weight$ remains unchanged. Because of the overlaps, deactivation of an interval member can activate another member, which in turn can deactivate the following member of the interval. This effect propagates until the first nonoverlapping member. If the number of activations/deactivations is even, then the weight remains the same, and the interval remains in its position in the *IBW* array. To determine the activation/deactivation parity, we use the sequential search, which we have discovered to be the best practical solution. However, solely for theoretical reasons, in the Appendix, we describe a formally more efficient solution that we use to improve bounds in the worst-case proof presented in Section V.

The active interval members in the affected area are not deactivated in the cases A1 and A4, when $I_A.weight$ is reduced by $I_{top}.lcp - 1$. In these two cases, a member of I_{top} is completely included into a member of I_A, and the weight of the affected interval changes only by the length $I_{top}.lcp$ of the shorter substring, minus the cost of storing the rule symbol.

After the replacement of all active substrings belonging to I_{top}, I_{top} is removed, and the new interval with the currently largest weight is at the top of the *IBW* array. The process continues until there are intervals in the *IBW*. Output of the compressed string and the rules is performed at the end to include all rule updates. The definitions of the rules are associated with the first (leftmost) occurrence of a substring replaced with a rule.

1) INCLUSION OF RULES INTO RULES

The existing rules can be included in a new rule, and a new rule can be formed as a part of an existing rule. Case A4 in Fig. 3b) illustrates a possible longer rule that completely includes the current rule. Case A3 in Fig. 3a) illustrates a possible rule within the current rule that can emerge later in the processing. If that occurs, the new rule must be incorporated into the previous longer rule. During the parsing, the information about the potential rules within existing rules is associated with the first occurrence of an interval member.

To perform a correct bookkeeping for the possibility of forming a new rule within the existing rule, we need to adapt the weight update process. When a new rule is formed from a substring of an existing rule, the affected intervals are limited only to those intervals whose members are shorter than the existing rule. Considering the example in Fig. 4, upon the formation of the *new* rule, only the weight of interval I_B is updated. The weight of interval I_A is already updated to a correct value upon the formation of the *previous* rule.

2) EMERGENCE OF NEW INTERVALS

In addition to updating the weights of the existing intervals, we must allow for the emergence of new intervals that may be created as a consequence of the shortening of the repeated substrings. The shortening can occur in the left affected

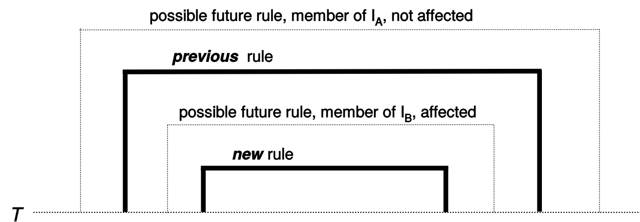


FIGURE 4. An example of the formation of a new rule inside the existing rule. Only the weight of interval I_B is updated.

neighborhood when the prefix of a substring that is a member of the affected interval is not a member of any existing interval, as illustrated in Fig. 3c). A new interval is formed if the same shortened substring occurs at least twice. The first candidates for the new intervals are stored separately and associated with their original intervals. If a second candidate occurs, a new interval is formed and placed in the appropriate position in the LCP-interval hierarchy. This approach consists of inserting the new interval in IBW in $O(1)$ time and updating LCP-interval tree in $O(\alpha)$ time.

IV. THE COMPLEXITY ANALYSIS AND EXPERIMENTAL RESULTS

The preprocessing involves the construction of the SA, SA^{-1}, LCP array, LCP-interval tree, and $IntervalMember$ and $InversePositions$ arrays. The SA can be constructed in $O(|T|)$ time [24] along with the SA^{-1} and the LCP array [26]. The LCP-interval tree data are derived in $O(S)$ time from the suffix and LCP arrays. The $IntervalMember$ and $InversePositions$ arrays are constructed in $O(S)$ time from the LCP-interval tree data and SA^{-1}, as presented in Algorithm 1. The summing of the initial weights and sorting of the IBW array has $O(S+|T|)$ complexity. Therefore, the total preprocessing time complexity is $O(S + |T|)$ or $O((\alpha + 1)|T|)$.

In the main parsing procedure, at each replacement of a substring in T with a rule, we have to visit each position p in T that belongs to the replaced substring and to the affected left neighborhood. The average number of the visited positions per substring replacement is bounded by the average LCP length lcp_{avg}, and the average number of interval members starting at each p is bounded by α. Since with each substitution of a substring with a rule at least one symbol is lost, there can be at most $|T|$ substitutions. Therefore, the potential number of weight updates is bounded by $(\alpha|T|lcp_{avg})$.

The weight update procedure comprises two more complex operations: the first is the verification whether a deactivation of an interval member changes the total weight or not, and the second is the repositioning of the interval in the (IBW) array. To verify whether another member of an interval becomes active, we use the sequential search and $InversePositions$ array to determine the consecutive positions of interval members. With the array implementation that we use, moving the interval in the IBW is performed in a constant time. Let $overlap_{avg}$ denote the average number of overlapped instances of a same substring starting at a position in T. Then, the total time complexity of our method in the average case is $O(\alpha|T|lcp_{avg}overlap_{avg})$.

In the worst case, we deal with the maximal sizes of α_{max}, lcp_{max} and $overlap_{max}$ that are all $O(|T|)$, which would lead to $O(N^4)$ complexity. This is amortized to $O(N^3)$ since the overlaps are checked only at the deactivation of an interval member, and there are at most $O(S = \alpha|T|)$ members of intervals. The worst case occurs in a pathological situation when the alphabet consists of only one symbol. Then, the probability of that symbol being the next character is always 1. For the real data, the distribution of symbol probabilities is more normal. In such a case, some results on the expected sizes of α, lcp_{avg} and $overlap_{avg}$ exist in the literature, although mostly for the Bernoulli model of a source. Except for the nodes at a depth of one, α is equivalent to the average depth of a suffix tree, which is in [27] shown to be $O(\log N/h)$, where h is the entropy of the source. Within the same model, lcp_{avg} is known to be $O(\log N)$ [28]. On the other hand, the size of $overlap_{avg}$ appears to be an open problem. To our knowledge, a result regarding the expected number of repeated substrings that overlap does not exist. However, based on the results for the number of runs and repetitions in a string (see, for example, [29] and [30]), it appears that the number of overlapped instances of a same substring, in a string produced by a Bernoulli source, tends to be a constant. This is supported by the analysis of the real-life data; e.g., for the DNA and the English texts, the values oscillate around 2 and 5, respectively, regardless of the input size.

Based on all of the above, we conjecture that the expected time complexity of our algorithm on "nonpathological" strings is $O(N \log^2 N)$, which is in accordance with the experimental results.

For our experiments, we have used the Pizza&Chili Corpus [31], the standard benchmark corpus for data compression that covers a very wide variety of different data types. Fig. 5 presents the processing times for six data sets from Pizza&Chili Corpus; *dna*, *proteins* and *english* from the main text corpus and *tm*, *dna_rep* and *ecoli* from the artificial, pseudo-real and real parts of the repetitive corpus, respectively. The results are obtained on a 3.6 GHz Xeon Gold 5122 processor with 256 GB RAM using a C++11 implementation of our method that can be downloaded from *https://bitbucket.org/marrose/esa-laf* compiled with the -O2 option. The discontinuities occur when lcp_{avg} radically changes. The presented examples are representative of the graphs and complexities obtained for every other data set from Pizza&Chili and Canterbury[2] corpora.

Fig. 6 and Table 1 illustrate the difference in performance between our algorithm, based on the local index updates, and the previous method, based on the reconstruction of the whole index. As an example, the differences in time complexity on two typical real data sets are presented in Fig. 6 for the first 10 MB of the *dna* file from the main corpus and *sources* file from the repetitive corpus. Table 1 presents the times needed to process the first 10 MB of all file types in the Pizza&Chili Corpus, both with the updates and the

[2]The older data compression benchmark [32].

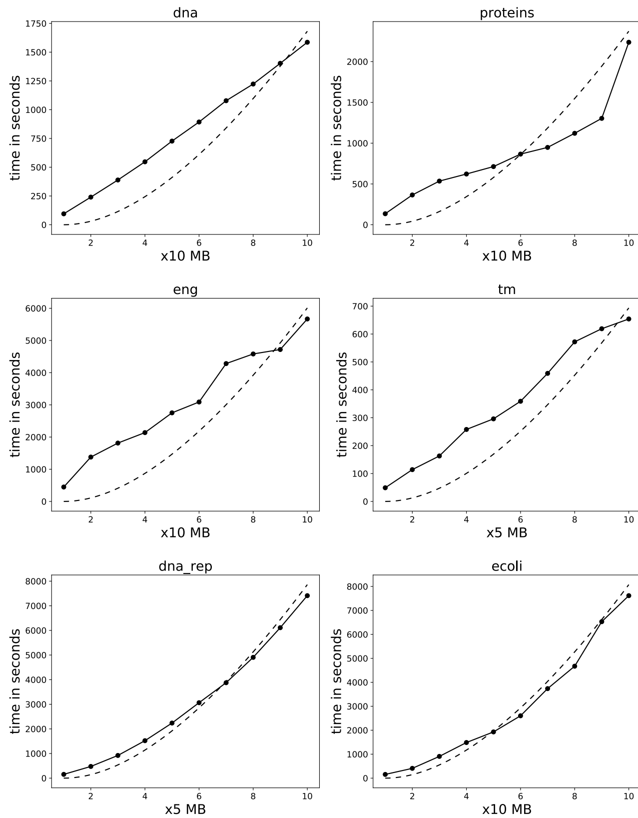


FIGURE 5. Processing times for the uniform subsets of the selected Pizza&Chili Corpus files. The dashed lines represent $N\log^2 N$ curves.

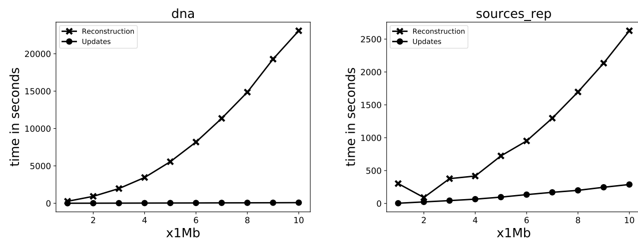


FIGURE 6. Compared execution times for the reconstruction and the updates methods for 10 MB subsets of two files from Pizza&Chili Corpus. sources_rep is a part of the *sources* file from the pseudo-real repetitive part of the corpus.

reconstruction methods. The time required for the processing of larger inputs with the reconstruction method was prohibitive (the processing of 100 MB inputs would require weeks).

The corresponding values of α and lcp_{avg} are also included in Table 1. The only cases where the reconstruction method is faster than the updates are the three files from the artificial part and one file from the pseudo-real part of the repetitive corpus. These four files all have very large values of lcp_{avg} that are comparable to the size of the sample itself. Those are the cases when the samples consist of a few repetitions of very long substrings, and the artificial repetitive part is the extreme part of the Pizza&Chili Corpus. For every other data set, with the more realistic values of lcp_{avg}, the updates method is faster. On the processed 10 MB samples, the factor by which the updates method is faster ranges from 1.85 for

TABLE 1. Processing times for 10 MB subsets of all files in the Pizza&Chili Corpus. Where not stated otherwise, values are given in seconds. M denotes the main corpus; AR, PR and RR denote the artificial, pseudo-real and real parts of the repetitive corpus, respectively; and w_l denotes *world_leaders* data set.

	data set	processing time: updates	processing time: reconstruct.	α	lcp_{avg}
M	sources	**298**	2931	14.46	929
	pitches	**65**	1461 min	11.17	207
	proteins	**106**	685 min	11.23	226
	dna	**82**	383 min	11.74	14
	english	**367**	6200	8.37	2348
	XML	**32**	8490	9.07	39
AR	fib	91	**13**	35.50	2549400
	rs	101	**15**	31.44	1854960
	tm	114	**15**	27.31	1433050
PR	dblp.xml	485	**123**	14.15	83618
	dna	**458**	1578	14.70	893
	english	**462**	2655	10.41	891
	proteins	**585**	3299	13.13	942
	sources	**300**	2627	14.15	929
RR	E_Coli	**123**	205 min	11.26	75
	cere	**344**	357 min	361.07	365
	coreutils	**84**	207 min	9.39	299
	einstein.de	**100**	198	15.66	14447
	einstein.en	**91**	168	16.23	16461
	influenza	**335**	2941	24.54	381
	kernel	**354**	1690	8.88	4307
	para	**178**	379 min	131.00	134
	w_l	**306**	2341	17.66	724

the *einstein.en* file to 2138 for the *pitches* file. Obviously, for larger files, the reconstruction method would not be feasible for most of the data sets.

It should be mentioned that we have not implemented the reconstruction procedure with the MAST structure but with a suffix array. We have used the code for SA construction from [24], modified for large alphabets, which is possibly somewhat faster than the suffix tree construction. With SA, the time needed to determine the largest area in each iteration is $O(N\alpha)$, as opposed to $O(N \log N)$ with MAST, which is asymptotically the same. When the value of α is comparatively large, it is possible that some corrective factor should be taken in the account. However, overall we do not believe that any reconstruction method based on ST would be much, if at all, faster than our implementation used in this experiment.

A. PRACTICALITY FOR LARGE INPUTS

The space requirement of our implementation is $(84 + 13\alpha)$ bytes of memory per input character. To demonstrate the feasibility of the new algorithm for processing large inputs, we have performed experiments on a human genome DNA sequence. We have used the human genome sequence file downloaded from [33] and cleaned of "NNN" triplets. The rest of the "N" characters (characters denoting unknown bases) are left to preserve the reading frame. In the original file, long runs of the "N" character increase α to a

few thousand, which is an impractical level for such large inputs. After the cleaning, the remaining file size is 3 GB, and $\alpha = 17.07$. With our software, we have been able to process the 850 MB prefix ($\alpha = 15.62$) of the genome on a 256 GB RAM machine in 3 hours and 40 minutes. Therefore, the whole genome can be processed with a TB of RAM in approximately 15 hours.

It should be noted that our current implementation cannot process files larger than 2 GB and that processing inputs larger than 4 GB would involve increasing the space consumption per character ratio. However, even if the space consumption is doubled, which is a very loose bound, processing of inputs of almost any size could be performed on a disk in a matter of weeks. In comparison, extrapolating from the data presented in Fig. 6, processing the human genome with the previous method would require decades.

B. VERIFICATION OF CORRECTNESS

The central idea of our method is to enumerate all possible changes in a string that can occur when a substring is replaced with a rule. Since it is difficult to formally prove that our algorithm indeed accounts for all possibilities, we have included the tester code for the verification of a single case, or a batch of random strings, in our published software. The tester uses brute force to test whether the choice of the next interval at every stage of the process belongs to the set of the correct choices. With this code, we have successfully tested the processing of random files of up to 3 MB.

V. THE HYBRID ALGORITHM AND THE IMPROVED WORST-CASE COMPLEXITY

A. ESTIMATE OF "PATHOLOGY"

In the context of our algorithm the "pathology" of a string is related to the size of lcp_{avg} compared to the expected $log_{|A|}|T|$ value. Our method is fast when lcp_{avg} is small compared to the length of the string. We consider an input to be a pathological input when the lcp_{avg} is of the same order of magnitude as $|T|$. Such strings almost never occur naturally and are often informally referred to in the literature as the pathological inputs.

Intuitively, a large lcp_{avg} implies that a few initial iterations of the LAF parsing must considerably reduce the size of the string. Since the complexity of one iteration with the reconstruction method is not dependent on the lcp_{avg}, the reconstruction must work faster than updates on strings with large lcp_{avg}. We illustrate this with the results of a simple experiment presented in Fig. 7.

We have generated random binary strings of 10^6 characters with a variable probability of symbols and measured the time needed for processing the strings with our algorithm and with the reconstruction method, averaged over five iterations. The obtained times are presented in Fig. 7 for a range of probabilities of a specific symbol from zero to 50%. The crossover value is at 4%. Therefore, we can observe that a binary string with length of 10^6 characters is pathological in the context of our method if the probabilities of two symbols are $p_1 \leq 4\%$

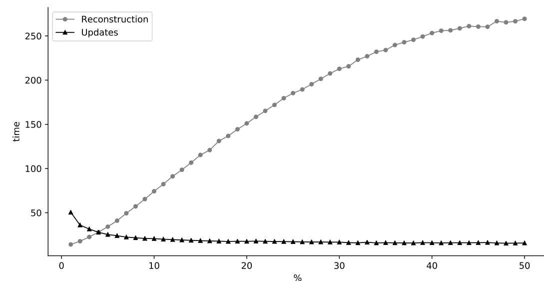


FIGURE 7. Processing times in seconds for different percentages of one symbol occurrences in randomly generated binary strings with lengths of 10^6 characters.

and $p_2 \geq 96\%$,[3] and for such strings, the reconstruction method works much faster than updates. This finding gives rise to the possibility of employing the hybrid approach that combines both methods.

The *hybrid algorithm* then involves the following:

- start with the reconstruction method and
- switch to the updates method when lcp_{avg} falls to the appropriate value.

We have used the hybrid algorithm to improve on the worst-case complexity of LAF parsing, and we give the proof in the next subsection. We do not, however, consider the hybrid algorithm as practical for real-life applications, except in cases when pathological inputs cannot be excluded. Considering the results on the Pizza&Chili data sets, a quick rule of thumb could be that a string can be efficiently processed with the updates method if its lcp_{avg} is within two orders of magnitude greater than the expected $log_{|A|}|T|$ value. The value of lcp_{avg} is computed in linear time during the initialization of data structures, and therefore, it can be used as a quick test to indicate how to continue with the processing.

B. WORST-CASE PROOF

At the beginning, we determine the worst-case complexity for updates method.

Lemma 2: The verification of the number of overlapped substrings in the updates algorithm can be performed in $O(\log N)$ time.

Proof: We give the appropriate algorithm in the Appendix. □

Therefore, the complexity of the updates algorithm in the worst case is $O(lcp_{max}\alpha_{max}N \log N) = O(lcp^2_{max} N \log N)$. The reconstruction algorithm has a complexity of $O(N \log N)$ per iteration for a total of $O(N^2 \log N)$ in cases where the number of iterations grows linearly with N. We consider a hybrid algorithm where we employ the reconstruction method while lcp_{max} is larger than a *boundary value B* and then switch to the updates algorithm when $lcp_{max} \leq B$. The complexity of the first phase is then

$$O(\mathcal{I}_B N \log N) \tag{1}$$

where $\mathcal{I}_B$ is the number of iterations of reconstruction required to satisfy the boundary condition, and the

[3]These values change with the string length; e.g., for the lengths of 10^5 and 10^7 characters, the crossover values are at 7% and under 2%, respectively.

complexity of the updates phase is

$$O(B^2 N \log N) \tag{2}$$

We use this model to prove the worst-case bound for the hybrid algorithm.

First, we need to formally establish that a reduction in the size of the string with each rule is at least linear with the value of lcp_{max}. We do so through the following lemma.

Lemma 3: Any compressible string is compressed by at least $\frac{lcp_{max}}{4}$ with one rule.

Proof:

Let $\mathcal{S}$ be a repeated substring such that $|\mathcal{S}| = lcp_{max}$. Obviously, the largest area replaced with the rule at any point in execution of the algorithm cannot be smaller than that covered with $\mathcal{S}$. Consider any two instances of $\mathcal{S}$, $\mathcal{S}_i$ and $\mathcal{S}_j$, at positions i and j, $i < j$. There are three possible cases:

1) $\mathcal{S}_i$ and $\mathcal{S}_j$ do not overlap. In this case, we can use $\mathcal{S}$ as a rule, and we obtain a compression of at least $|\mathcal{S}| - 1$.
2) $\mathcal{S}_i$ and $\mathcal{S}_j$ overlap, and $j > i + \frac{|\mathcal{S}|}{2}$. The first half, or more, of $\mathcal{S}$ is replaced with a rule, and the compression is then at least $\frac{|\mathcal{S}|}{2} - 1$.
3) $\mathcal{S}_i$ and $\mathcal{S}_j$ overlap, and $j \leq i + \frac{|\mathcal{S}|}{2}$. $\mathcal{S}$ is then a periodic string with period $j - i$. The rule composed of the first $j - i$ characters compresses the string by at least $(j - i - 1)(\lfloor \frac{|\mathcal{S}|/2}{j-i} \rfloor)$. When $j - i \geq 2$, this is trivially bounded below by $\frac{|\mathcal{S}|}{4}$. When $j - i = 1$, i.e., $\mathcal{S}$ is a sequence of a same symbol, this case degenerates to the first case with $|\mathcal{S}| = \frac{|\mathcal{S}|}{2}$.

Since $|\mathcal{S}| = lcp_{max}$, the lemma is proved. □

Next, we prove an important result on $\mathcal{I}_B$.

Lemma 4: The number of iterations $\mathcal{I}_B$ necessary to reduce lcp_{avg} to the value of B is not larger than $4\frac{N}{B}$.

Proof:

Let $\mathcal{W}_i$ denote the net gain of the i-th rule, $\mathcal{L}_i$ denote the lcp_{avg} before the i-th rule replacement and N_i denote the length of the compressed string and all the rules before the i-th rule replacement. Then, $N_{i+1} = N_i - \mathcal{W}_i$.

Let us assume that $\mathcal{I}_B > 4\frac{N}{B}$. Then, $N_{\mathcal{I}_B} < N_{4\frac{N}{B}}$, and for every $k < 4\frac{N}{B}$ must hold $\mathcal{L}_k > B$. By Lemma 2, we have $\mathcal{W}_k \geq \frac{B}{4}$. Then:

$$N_{4\frac{N}{B}} = N - \sum_{j=1}^{4\frac{N}{B}} \mathcal{W}_j \leq N - 4\frac{N}{B}\frac{B}{4} = 0$$

Since this is impossible, it follows that $\mathcal{I}_B \leq 4\frac{N}{B}$. □

Finally, we prove the new worst-case bounds for greedy parsing.

Theorem 1: Using the hybrid method, the complexity of the LAF parsing of a string is $O(N^{\frac{5}{3}} \log N)$.

Proof:

From (1) and Lemma 2, we have the complexity for the reconstruction phase of the hybrid algorithm as

$$O(\frac{N^2}{B} logN) \tag{3}$$

Setting $B = \sqrt[3]{N}$ in (2) and (3), we obtain the complexity of both phases of the algorithm as $O(N^{\frac{5}{3}} \log N)$. □

VI. CONCLUSION

We present the first feasible method for the largest area parsing of a string that works fast on real data sets (represented with the main and the real repetitive parts of the Pizza&Chili Corpus). The previous method, which is about twenty years old, cannot realistically be used for files larger than a few MB. Since the grammar compressed strings have advantages with compressed pattern matching and querying and since the LAF parsing produces the best results overall of all grammar based strategies, a practical LAF parsing method can be of interest to the researchers and practitioners in the field. Our algorithm is also a contribution to the research on the important subject of dynamic indexing.

In the theoretical domain, we have improved the worst-case bound for LAF parsing from $O(N^2 \log N)$ to $O(N^{\frac{5}{3}} \log N)$. This is achieved using the hybrid between our algorithm, based on index updates, and the previous algorithm, based on index reconstruction.

Although the guiding idea for the new LAF algorithm was found in the previous work on the longest first parsing [22], the new algorithm does not follow trivially from that in [22]. The mechanics of the LAF algorithm are distinct, and the complexity includes the additional lcp_{avg} factor. The code for an implementation of our LAF parsing method is publicly available at *https://bitbucket.org/marrose/esa-laf*.

VII. APPENDIX: COMPLEXITY OF THE OVERLAP PROBLEM

We show that the overlap problem can be solved within an additional factor of $O(\log N)$ operations in the worst case.

A. PROBLEM DEFINITION

Given the set of m instances $\mathcal{S}_i$, $i = 1, 2, .., m$ at positions p_i of a repeated substring $\mathcal{S}$ of length $|\mathcal{S}|$, we have to maintain information about the size of the largest subset of nonoverlapping instances while supporting removals of instances from the set.

B. SOLUTION

Let $f(i, j)$ be the size of the largest subsequence of nonoverlapping instances of $\mathcal{S}$ inclusive between the positions p_i and p_j, i.e., subsequence with no two positions having a difference smaller than $|\mathcal{S}|$. Let $next(i)$ return the index of the first nonoverlapped instance (where it exists) following the instance at the position p_i. Then, $f(i, j)$ is equal to the smallest positive integer k such that $next^k(i) > j$.

To efficiently compute f, we use a static table J, where $J_{i,k} = next^{(2^k)}(i)$. Table J has $O(NlogN)$ elements and can be computed in $O(NlogN)$ time using the following expression:

$$J_{i,k} = \begin{cases} next(i) & \text{if } k = 0 \\ J_{J_{i,k-1},k-1} & \text{otherwise} \end{cases}$$

Algorithm 2 Computing f(i, j)

```
  Input: i, j
  Result: f(i, j)
1 result:= 1;
2 for k ← ⌊log2 N⌋ to 0 by −1 do
3     if J[i][k] ≤ j then
4         result ← result + 2^k;
5         i ← J[i][k];
6 end
7 return result
```

Table J allows us to compute values of f in $O(logN)$ time using Algorithm 2 to essentially perform a binary search for the answer.

We recall that during the course of our LAF parsing algorithm, instances of the repeated substrings are eliminated when they partially overlap with the new rules. We say that eliminated instances are "dead" or "killed". Initially, all instances are "alive". When an instance is killed, we need to verify whether this actually changes the weight of the interval due to overlaps. In contrast from our practical code, here, in this theoretical construct, we do not use the active/inactive status of the individual substring instances to determine the effect of the overlaps but instead recalculate the value of f for each affected interval after the formation of a new rule.

The relevant changes can occur within chains of overlapping alive instances of a repeated substring. When a rule is formed within a chain, there are two possible cases.

1) *A rule divides a chain into two new chains.* This covers the cases A2, A3, and A5 in Fig. 3. Then, except at the start or the end of a chain, more than one repeated substring instance is killed, and the dead instances always divide the original chain into two distinct new chains of live instances.

   To process this case, we maintain two items:
   - set C of $[start, end]$ pairs, representing chains of currently alive instances and
   - the cumulative weight of the interval $r = \sum_{c \in C} f(c)$.

   Initially, $C = \{[1, N]\}$ and $r = f(1, N)$.

   Let us denote with a and b the first and the last instance in a group of killed instances, respectively. At each position of a new rule, we need to perform the following four steps.
   a) Find the chain in C that contains subchain $[a, b]$. Let that be chain $[x, y]$.
   b) Remove $[x, y]$ from C and subtract $f(x, y)$ from r.
   c) If $x < a$, add $[x, a-1]$ to C and add $f(x, a-1)$ to r.
   d) If $b < y$, add $[b+1, y]$ to C and add $f(b+1, y)$ to r.

   If C is implemented as a balanced binary search tree, then all four steps are performed in $O(logN)$ time. This process yields the total complexity of $O(NlogN)$ as the number of iterations is bound by N since each iteration kills at least one live instance.
2) *A rule does not break the chain of live instances.* This occurs with the cases A1 and A4 in Fig. 3. In this case, we cannot split chains in C, and we need to recalculate J. However, now, all instances of the affected substring are visited during the replacement of I_{top} with a rule. This means that when we recalculate J in $O(\log N)$ time for every instance, the total cost is amortized within $O(N \log N)$ bound.

This result is mainly theoretical since its implementation would require considerable additional memory space and, based on the corpora data, would be slower than the sequential search in most practical cases. Therefore, we do not recommend this as a practical solution.

ACKNOWLEDGMENT

The author would like to thank the authors of [24] for using their suffix array construction code in our software and the anonymous reviewers of the previous versions of this article for their useful comments and suggestions.

REFERENCES

[1] J. C. Kieffer and E.-H. Yang, "Grammar-based codes: A new class of universal lossless source codes," *IEEE Trans. Inf. Theory*, vol. 46, no. 3, pp. 737–754, May 2000.
[2] C. G. Nevill-Manning and I. H. Witten, "On-line and off-line heuristics for inferring hierarchies of repetitions in sequences," *Proc. IEEE*, vol. 88, no. 11, pp. 1745–1755, Nov. 2000.
[3] M. Lohrey, "Algorithmics on SLP-compressed strings: A survey," *Groups Complex. Cryptol.*, vol. 4, no. 2, pp. 241–299, Jan. 2012.
[4] F. Claude and G. Navarro, "Improved grammar-based compressed indexes," in *Proc. SPIRE*, vol. 7608, L. Calderón-Benavides, C. González-Caro, and E. C. N. Ziviani, Eds. Berlin, Germany: Springer, 2012, pp. 180–192.
[5] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.
[6] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Trans. Inf. Theory*, vol. 51, no. 7, pp. 2554–2576, Jul. 2005.
[7] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M. L. Schmid, "On the complexity of grammar-based compression over fixed alphabets," in *Proc. ICALP*, vol. 2016, pp. 1–14.
[8] A. Jez, "A really simple approximation of smallest grammar," in *Proc. CPM*, vol. 8486, LNCS, A. S. Kulikov, S. O. Kuznetsov, and P. A. Pevzner, Eds. Cham, Switzerland: Springer, 2014, pp. 182–191.
[9] R. Nakamura, S. Inenaga, H. Bannai, T. Funamoto, M. Takeda, and A. Shinohara, "Linear-time text compression by longest-first substitution," *Algorithms*, vol. 2, no. 4, pp. 1429–1448, Nov. 2009.
[10] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proc. IEEE*, vol. 88, no. 11, pp. 1722–1732, Nov. 2000.
[11] J. Bentley and D. McIlroy, "Data compression using long common strings," in *Proc. DCC Data Compress. Conf.*, 1999, pp. 287–295.
[12] R. Nakamura, H. Bannai, S. Inenaga, and M. Takeda, "Simple linear-time off-line text compression by longest-first substitution," in *Proc. Data Compress. Conf. (DCC)*, 2007, pp. 123–132.
[13] A. Apostolico and S. Lonardi, "Off-line compression by greedy textual substitution," *Proc. IEEE*, vol. 88, no. 11, pp. 1733–1744, Nov. 2000.
[14] G. S. Brodal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen, "Solving the string statistics problem in time o(n log n)," in *Proc. ICALP*, vol. 2002, pp. 728–739.
[15] R. Carrascosa, F. Coste, M. Gallé, and G. Infante-Lopez, "Searching for smallest grammars on large sequences and application to DNA," *J. Discrete Algorithms*, vol. 11, pp. 62–72, Feb. 2012.
[16] K. J. Conrad and P. R. Wilson, "Grammatical ziv-lempel compression: Achieving PPM-class text compression ratios with LZ-class decompression speed," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2016, p. 586.

[17] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge, U.K.: Cambridge Univ. Press, 1997.
[18] H. Cohen and E. Porat, "Range non-overlapping indexing," in *Algorithms and Computation*, vol. 5878, Y. Dong, D.-Z. Du, and O. Ibarra, Eds. Berlin, Germany: Springer, 2009, pp. 1044–1053.
[19] M. Gallé, P. Peterlongo, and F. Coste, "In-place update of suffix array while recoding words," in *Proc. Stringology*, J. Holub and J. Zdárek, Eds., 2008, pp. 54–67.
[20] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard, "Dynamic extended suffix arrays," *J. Discrete Algorithms*, vol. 8, no. 2, pp. 241–257, Jun. 2010.
[21] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, Mar. 2004.
[22] S. Ristov and D. Korenčić, "Using static suffix array in dynamic application: Case of text compression by longest first substitution," *Inf. Process. Lett.*, vol. 115, no. 2, pp. 175–181, Feb. 2015.
[23] S. Ristov and D. Korenčić, "Fast construction of space-optimized recursive automaton," *Softw., Pract. Exper.*, vol. 45, no. 6, pp. 783–799, Jun. 2015.
[24] G. Nong, S. Zhang, and W. Hong Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Trans. Comput.*, vol. 60, no. 10, pp. 1471–1484, Oct. 2011.
[25] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. CPM*, 2001, pp. 181–192.
[26] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Proc. CPM*, 2009, pp. 181–192.
[27] W. Szpankowski, "A generalized suffix tree and its (Un)Expected asymptotic behaviors," *SIAM J. Comput.*, vol. 22, no. 6, pp. 1176–1198, Dec. 1993.
[28] A. Apostolico and W. Szpankowski, "Self-alignments in words and their applications," *J. Algorithms*, vol. 13, no. 3, pp. 446–467, Sep. 1992.
[29] S. J. Puglisi, J. Simpson, and W. F. Smyth, "How many runs can a string contain?" *Theor. Comput. Sci.*, vol. 401, nos. 1–3, pp. 165–171, Jul. 2008.
[30] M. Christodoulakis, M. Christou, M. Crochemore, and C. S. Iliopoulos, "Overlapping factors in words," *Australas. J. Combinatorics*, vol. 57, pp. 49–64, Oct. 2013.
[31] *Pizza & Chili Corpus*. Accessed: Oct. 28, 2017. [Online]. Available: http://pizzachili.dcc.uchile.cl/index.html
[32] *Canterbury Corpus*. Accessed: Sep. 13, 2015. [Online]. Available: http://corpus.canterbury.ac.nz
[33] *Human Genome*. Accessed: Mar. 14, 2020. [Online]. Available: ftp://ftp.ensembl.org/pub/release-99/fasta/homo_sapiens/dna/Homo_sapiens.GRCh38.dna.primary_assembly.fa.gz

IVAN KATANIĆ was born in Croatia, in 1993. He received the B.S. and M.S. degrees in computer science from the University of Zagreb, in 2017.

From 2017 to 2019, he was a Research Engineer at Blue Vision Labs. Since 2019, he has been a Quantitative Developer at Jump Trading, London, U.K. He is a coauthor of one conference paper in the field of string algorithms. His research interests include string algorithms and data structures. He won silver medals at the ACM ICPC competition, in 2014 and 2015.

STRAHIL RISTOV was born in Zagreb, Croatia, in 1959. He received the B.S. degree in electrical engineering and the M.S. and Ph.D. degrees in computer science from the University of Zagreb, in 1997.

Since 1990, he has been a Researcher at the Department of Electronics, Ruer Bošković Institute, Zagreb, where he currently holds a senior associate scientist position. He is the Head of the Laboratory for Information and Signal Processing. He is the author or coauthor of 30 journal or conference papers. His research interests include string algorithms, data compression, and algorithms in bioinformatics and population genetics.

MARTIN ROSENZWEIG was born in Zagreb, Croatia, in 1997. He received the B.S. degree in mathematics from the University of Zagreb, in 2019. He is currently pursuing the M.S. degree in mathematics and data science at Technische Universität München, Germany.

He is a coauthor of one journal article in the field of bioinformatics. His research interests include string algorithms and data structures.