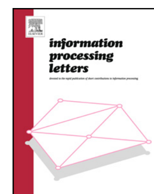
Information Processing Letters 115 (2015) 175–181

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl



Using static suffix array in dynamic application: Case of text compression by longest first substitution

Strahil Ristov *, Damir Korenčić

Ruđer Bošković Institute, Department of Electronics, Bijenička 54, 10000 Zagreb, Croatia

A R T I C L E I N F O

Article history:
Received 7 May 2014
Received in revised form 11 July 2014
Accepted 22 August 2014
Available online 16 September 2014
Communicated by M. Chrobak

Keywords:
Algorithms
Enhanced suffix array
Grammar text compression
Longest first substitution
Text index update

A B S T R A C T

Over the last decade, enhanced suffix arrays (ESA) have replaced suffix trees in many applications. Algorithms based on ESAs require less space, while allowing the same time efficiency as those based on suffix trees. However, this is only true when a suffix structure is used as a static index. Suffix trees can be updated faster than suffix arrays, which is a clear advantage in applications that require dynamic indexing. We show that for some dynamic applications a suffix array and the derived LCP-interval tree can be used in such a way that the actual index updates are not necessary. We demonstrate this in the case of grammar text compression with longest first substitution and provide the source code. The proposed algorithm has $O(N^2)$ worst case time complexity but runs in $O(N)$ time in practice.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

It was shown in [1] that string algorithms based on a suffix array (*SA*) enhanced with additional data structures are equivalent regarding the execution time complexity to those based on a suffix tree (*ST*), while requiring less space. The suffix array can be enhanced with a choice of additional data, such as the inverted suffix array, longest common prefix (LCP) array, suffix link table, LCP-interval tree, and interval child table. There exist a number of efficient algorithms for SA construction, both theoretically linear and practically fast [2,3]. Efficient linear-time algorithms also exist for the construction of the LCP array [4,5], and all of the other data are easily derived from a suffix array, or during its construction [1]. As a result, suffix arrays has been successfully used to replace suffix trees in various applications [6,7]. This is, however, only true for static applications. In cases when a dynamic update is needed the suffix tree still has an advantage. A typical example of a dynamic application is text compression by substitution of a repeated substring with a new symbol or with a pointer, either to a previous occurrence in the string itself or to an entry in an external dictionary. This poses a problem of effective dynamic text indexing that has been addressed mostly by modifications in ST construction. Dynamic text indexing solves index updates after localized modifications of the input string, i.e., after replacement of a single occurrence of a substring. The specific problem is a global index update: a simultaneous substitution of multiple occurrences of a substring that, in particular, arises in the field of grammar compression. So far, the only method for a global update in (amortized) linear time is based on the suffix tree [8].

A few papers have addressed the usage of suffix arrays for dynamic indexing [9–11], even in the context of the global updates [9], but their solutions do not attain linear time complexity. In [11] authors show that the number of steps necessary for an update of a suffix array is proportional to the average LCP value LCP_{ave}. Since LCP_{ave} is much smaller than the size of the text, updating SA is much faster than rebuilding it from scratch. A fully dynamic index must support random access insertions and deletions

* Corresponding author.
E-mail addresses: ristov@irb.hr (S. Ristov), damir.korencic@irb.hr (D. Korenčić).

http://dx.doi.org/10.1016/j.ipl.2014.08.014
0020-0190/© 2014 Elsevier B.V. All rights reserved.

(with a substitution as a combination thereof) and the method presented in [11] supports both of them. In this paper we show that, for a restricted problem of substituting parts of a string in predefined order, it is possible to use the enhanced suffix array to perform global updates in practically linear time. This result is based on the fact that the sum of the depths of LCP-interval tree over all the positions in SA is in effect linear with respect to the size of the input string. We have previously explored this fact in the construction of a fast algorithm for the compression of finite automata, based on the replacement of a repeated substring with the pointer to a previous occurrence in the string [12]. Here we demonstrate how this can be used to solve another problem in string compression, specifically, grammar text compression with longest first substitution.

2. Grammar text compression

The concept of context free grammar production rules as a means of string compression was formalized in [13] and has been known as the grammar text compression since [14,15]. However, grammar text compression is only a specific approach to the problem of compression by textual substitution [16,17] that has been addressed earlier [17–19].

The idea of grammar compression is to find as small as possible a context free grammar (CFG) that uniquely produces the input string. Rules of such grammar can be further compressed with the appropriate codes, but that is another subject of research that we shall not discuss in this paper. Instead, we focus on the first part of the problem, finding the smallest CFG for a given string. A CFG is composed of production rules that replace repeated substrings in text T. The optimal assignment of substrings for the replacement is an NP-hard problem [16], as a result, various greedy heuristics have been proposed: longest substrings first [19], most frequent substrings first [20], and largest area first that finds the substrings that have the highest product of their length and the number of non-overlapping occurrences [17]. In the general case, better compression results can be achieved with most frequent first and largest area first variants [18]. The longest first approach is particularly successful in finding long distant repeats, which is advantageous in applications involving long DNA sequences [19]. The grammar compressed strings support efficient searches without decompression [14] and grammar compression gives insight into the hierarchical structure of the text [18]. So far, there does not exist a linear time algorithm for largest area first method, but such algorithms exist for most frequent first [20] and for longest first substitutions [8]. The algorithm in [8] is based on the suffix tree and is the only linear time algorithm for this task. In this paper we present an algorithm that is based on the suffix array and that runs in time that is quadratic in the worst case but is in practice linear with the input text size.

2.1. Grammar text compression with longest first substitution

The longest first method at each iteration i replaces the longest repeated substrings in T with a new rule R_i.

```
T: abcabcabdfabcabghabcabcab

R0: 1df2gh1
R1: 2c3
R2: 3c3
R3: ab
```

Fig. 1. An example of grammar compression with longest first substitution.

An example of the smallest CGF representing a text are the four rules in Fig. 1. R_0 represents the original text T, and the rest of the rules represent the repeated substrings, from the longest to the shortest. The repeated string that can be replaced with a rule must be at least two characters long, and must occur at least twice in the CFG. Otherwise the total size of the CFG could increase with the respect to the original string.

The example in Fig. 1 presents a CFG where the substrings repeated within rules are also replaced with the new rules. This variant of longest first substitution is referred to as LFS2 in [8], as opposed to LFS where rules are stored as the explicit substrings of T. Obviously, LFS2 variant provides more compression and this is the approach we address in this paper. The problem of longest first substitution is dynamic in the sense that at each iteration we have to find the longest repeated substring, replace its every occurrence with a new rule, and then recalculate the positions of the new longest repetitions in case that the performed replacements have interfered with other repetitions. This requires some sort of a dynamic index. If that index can be updated in constant time, we can perform the complete parsing of T in time linear with its length $|T|$. This is the approach taken in [8] where the authors show that their algorithm is the only one with truly linear time complexity. They achieve linearity by amortized constant time updates of nodes in a sparse lazy suffix tree. Updates of that sort are impossible in a suffix array in constant time, therefore we do not attempt to modify the array itself. Instead, we use two additional random access tables and base our algorithm on one observed property of the LCP-interval tree.

3. Longest first substitution using suffix array

Our algorithm uses a suffix array augmented with LCP-interval tree data. The suffix array and the corresponding LCP-interval tree for the text from the example on Fig. 1 are presented in Fig. 2. Nodes in LCP-interval tree represent intervals in SA that correspond to suffixes of T with the same LCP value. These intervals are marked in Fig. 2, but only for LCP values of 2 or more. These are the intervals we have to traverse during the execution of our algorithm.

In addition to SA and LCP-interval tree, we use two tables: *s_t* (substitution table), and *arp_t* (active rule position table), both with $|T|$ elements. *s_t* is used to store rule labels and denote the positions of characters in T that have been substituted with a rule. Each position in *s_t* corresponds to a position in T. Let us denote with *original* the otherwise unused value that we use to indicate that a character in T is not replaced with a rule. Similarly, with *replaced* we denote a value that we use to indicate that a character in T belongs to a substring that is replaced with

Download English Version:

https://daneshyari.com/en/article/10331882

Download Persian Version:

https://daneshyari.com/article/10331882

Daneshyari.com