Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 75–80
Information Processing Letters
www.elsevier.com/locate/ipl

Computing Longest Previous Factor in linear time and applications ☆

Maxime Crochemore [a,b,1], Lucian Ilie [c,*,2]

[a] *Department of Computer Science, King's College London, London WC2R 2LS, UK*
[b] *Institut Gaspard-Monge, Université Paris-Est, F-77454 Marne-la-Vallée Cedex 2, France*
[c] *Department of Computer Science, University of Western Ontario, N6A 5B7, London, Ontario, Canada*

Received 27 July 2007; received in revised form 16 October 2007; accepted 17 October 2007

Available online 23 October 2007

Communicated by L.A. Hemaspaandra

Abstract

We give two optimal linear-time algorithms for computing the Longest Previous Factor (LPF) array corresponding to a string w. For any position i in w, $\mathsf{LPF}[i]$ gives the length of the longest factor of w starting at position i that occurs previously in w. Several properties and applications of LPF are investigated. They include computing the Lempel–Ziv factorization of a string and detecting all repetitions (runs) in a string in linear time independently of the integer alphabet size.

Keywords: Design of algorithms; Analysis of algorithms; Strings; Suffix array; Longest common prefix; Longest previous factor; Lempel–Ziv factorization; Repetitions; Runs

1. Introduction

Given a string w, we introduce the Longest Previous Factor (LPF) array defined as follows. For any position i in w, $\mathsf{LPF}[i]$ gives the length of the longest factor of w starting at position i that occurs previously in w. Formally, if $w[i]$ denotes the ith letter of w and $w[i\,..\,j]$ is the factor $w[i]w[i+1]\dots w[j]$, then

$$\mathsf{LPF}[i] = \max\bigl(\bigl\{\ell \mid w[i\,..\,i+\ell-1] \text{ is a factor of } w[0\,..\,i+\ell-2]\bigr\} \cup \{0\}\bigr).$$

We give two linear-time (optimal) algorithms for computing LPF using suffix arrays. The first uses no additional information whereas the second uses the longest common prefix array which is often part of the suffix array data structure. Previously such algorithms involved computing the suffix trees, which are more complex and take a lot of space. Also, a logarithmic factor of the size of the alphabet often appears in the complexity. Our algorithms use suffix arrays, are much simpler, and their complexity is alphabet independent.

One important application is computing the Lempel–Ziv factorization [14]. Recently Abouelhoda et al. [1] gave a suffix-array-based algorithm for computing Lempel–Ziv factorization. However, their algorithm is

☆ This work has been presented at the AutoMathA'07 Conference, see [M. Crochemore, L. Ilie, Computing local periodicities in strings, invited talk, AutoMathA'07, Palermo, Italy, June 2007 [5]].

* Corresponding author.
E-mail addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), ilie@csd.uwo.ca (L. Ilie).

[1] Research supported in part by CNRS.
[2] Research supported in part by NSERC.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2007.10.006

i	SA[i]	LCP[i]	$\mathsf{suf}_{\mathsf{SA}[i]}$	$\mathsf{prev}_<[\mathsf{SA}[i]]$	$\mathsf{prev}_>[\mathsf{SA}[i]]$	LPF[SA[i]]	PrevOcc[SA[i]]
0	8	0	aaabab	−1	3	2	3
1	9	2	aabab	8	3	3	3
2	3	3	aabbbaaabab	−1	0	1	0
3	12	1	ab	3	10	2	10
4	10	2	abab	3	0	2	0
5	0	2	abbaabbbaaabab	−1	−1	0	−1
6	4	3	abbbaaabab	0	2	3	0
7	13	0	b	4	7	1	7
8	7	1	baaabab	4	2	3	2
9	2	3	baabbbaaabab	0	1	1	1
10	11	2	bab	2	6	2	2
11	6	1	bbaaabab	2	1	4	1
12	1	4	bbaabbbaaabab	0	−1	0	−1
13	5	2	bbbaaabab	1	−1	2	1

Fig. 1. The arrays SA, LCP, and LPF for the string abbaabbbaaabab.

essentially a simulation of the suffix tree using the suffix array. The description in [1] is very brief but it seems that their approach can be used to achieve similar goals with ours, nevertheless in a significantly more complicated way.

Simultaneously and independently of our work, Chen et al. [2] gave an algorithm that is similar with our second one. Our first algorithm is more general and our approach for the second gives a clearer explanation as well as more insight into the structure of LPF.

2. Suffix arrays

We recall in this section briefly the notions of suffix array and longest common prefix. Consider a string $w = w[0..n-1]$ of length n over an alphabet A that is an integer interval of size no more than n^c, for some constant c. The suffix of w starting at position i is denoted by $\mathsf{suf}_i = w[i..n-1]$, for $0 \leqslant i \leqslant n-1$. The *suffix array* of w, [16], denoted SA, gives the suffixes of w sorted ascendingly in lexicographical order, that is, $\mathsf{suf}_{\mathsf{SA}[0]} < \mathsf{suf}_{\mathsf{SA}[1]} < \cdots < \mathsf{suf}_{\mathsf{SA}[n-1]}$. The suffix array of the string abbaabbbaaabab is shown in the second column of Fig. 1.

Often the suffix array is used in combination with another array, the Longest Common Prefix (LCP) which gives the length of the longest common prefix between consecutive suffixes of SA, that is, LCP[i] is the length of the longest common prefix of $\mathsf{suf}_{\mathsf{SA}[i]}$ and $\mathsf{suf}_{\mathsf{SA}[i-1]}$; see the third column of Fig. 1 for an example.

3. A direct algorithm

We give first a direct algorithm for computing LPF from the suffix array. We compute also, for each i, a position PrevOcc[i] $< i$ where the longest previous factor at i occurs.[3] (If LPF[i] $= 0$, then PrevOcc[i] $= -1$.) Both arrays for our example are shown in the last two columns in Fig. 1.

The idea of the algorithm is as follows. For any position i, the longest factor starting at i that occurs also to the left of i in w is the longest common prefix between the suffix suf_i and the suffixes starting to the left of i in w, that is, suf_j, $0 \leqslant j \leqslant i-1$. However, given SA, we need only consider those which are closest to suf_i in SA. We shall therefore compute, for each i, the closest positions in SA that are smaller than i; in most cases there will be two such positions, one before and one after i in SA. Denote them by $\mathsf{prev}_<[i]$ and $\mathsf{prev}_>[i]$, respectively. If one of them does not exists, then we assign the value -1; see columns 5 and 6 in Fig. 1. Rephrasing the above, LPF[i] is obtained as the length of the longest common prefix between suf_i and either $\mathsf{suf}_{\mathsf{prev}_<[i]}$ or $\mathsf{suf}_{\mathsf{prev}_>[i]}$, whichever is longer.

After $\mathsf{prev}_<$ and $\mathsf{prev}_>$ are found, LPF is computed for all values of i in increasing order, using the property that $\mathsf{LPF}[i] \geqslant \mathsf{LPF}[i-1]-1$. Thus, we already know that $w[i..i+\mathsf{LPF}[i-1]-2]$ occurred to the left of i and need only try to extend it. A problem appears because we do not know whether we should compare suf_i to $\mathsf{suf}_{\mathsf{prev}_<[i]}$ or $\mathsf{suf}_{\mathsf{prev}_>[i]}$. We shall therefore compute two arrays, $\mathsf{LPF}_<[0..n-1]$ and $\mathsf{LPF}_>[0..n-1]$; they have the same meaning as LPF except that they consider only positions corresponding to suffixes lexicographically smaller, resp. larger, than suf_i. Formally, we have

[3] Note that a suffix-tree-based algorithm would compute the leftmost such position in the string whereas our algorithm might produce a different one. For instance, in our example, PrevOcc[12] $= 10$ but the left most occurrence of ab starts at 0.