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, 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.
© 2007 Elsevier B.V. All rights reserved.

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, 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 $w[i]w[i+1]\ldots w[j]$, 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 © 2007 Elsevier B.V. All rights reserved.
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.

```
COMPUTE_LPF(w, prev_<, prev_>)
LPF[0] ← LPF_<[0] ← LPF_>[0] ← 0
for i from 1 to n − 1 do
  j ← max(LPF_<[i − 1] − 1, 0);
  k ← max(LPF_>[i − 1] − 1, 0)
  if (prev_<[i] = −1) then LPF_<[i] ← 0
  else while (w[i + j] = w[prev_<[i] + j]) do j ← j + 1
      LPF_<[i] ← j
  if (prev_>[i] = −1) then LPF_>[i] ← 0
  else while (w[i + k] = w[prev_>[i] + k]) do k ← k + 1
      LPF_>[i] ← k
  LPF[i] ← max(LPF_<[i], LPF_>[i])
  if (LPF[i] = 0) then PrevOcc[i] ← −1
  else if (LPF_<[i] > LPF_>[i]) then PrevOcc[i] ← prev_<[i]
      else PrevOcc[i] ← prev_>[i]
return LPF and PrevOcc
```

Fig. 2. Algorithm for computing LPF directly from the SA via arrays $\mathsf{prev}_<$ and $\mathsf{prev}_>$.

$$\mathsf{LPF}_<[i] = \max\Big(\big\{j \mid \exists k < i, w[i\,..\,i+j-1] = w[k\,..\,k+j-1] \text{ and } \mathsf{suf}_k < \mathsf{suf}_i\big\} \cup \{0\}\Big)$$

and $\mathsf{LPF}_>[i]$ is defined identically except for the last condition which becomes $\mathsf{suf}_k > \mathsf{suf}_i$. Consider one of them, say $\mathsf{LPF}_<$. We still have that $\mathsf{LPF}_<[i] \geqslant \mathsf{LPF}_<[i-1]-1$. This is because, if $w[i-1] = w[k-1]$, then the order between suf_i and suf_k is the same as between suf_{i-1} and suf_{k-1}. That means, we already know that suf_i and $\mathsf{suf}_{\mathsf{prev}_<[i]}$ have a common prefix of length $\mathsf{LPF}_<[i-1]-1$ and we check only the following letters. This explains why our algorithm runs in $\mathcal{O}(n)$ time. Finally, $\mathsf{LPF}[i]$ is the maximum between $\mathsf{LPF}_<[i]$ and $\mathsf{LPF}_>[i]$. The algorithm is given in Fig. 2.

Computing the arrays $\mathsf{prev}_<$ and $\mathsf{prev}_>$ is a matter of manipulating data structures. We can construct a doubly linked list with the elements of SA and two extra elements at the beginning and at the end with value -1, for the case $\mathsf{prev}_<[i]$ or $\mathsf{prev}_>[i]$ do not exist. The values are computed in decreasing order of i, from n to 0, and each i is removed once the values for i have been computed. When i is considered, it is the largest left in the list and therefore the one before and the one after in the list will give the values of $\mathsf{prev}_<[i]$ and $\mathsf{prev}_>[i]$. In fact pointers can be avoided, and arrays used instead, because after the doubly linked list is created only deletions are performed. The details are omitted.

It should be clear from the above discussion that the arrays $\mathsf{prev}_<$ and $\mathsf{prev}_>$ can be computed in time $\mathcal{O}(n)$. The algorithm COMPUTE_LPF uses $\mathcal{O}(n)$ space. Using the fact that the suffix array of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time by any of the algorithms in [7,9,11,12], we obtain:

Theorem 1. *Given a string of length n over an integer alphabet, the* LPF *and* PrevOcc *arrays can be computed in time and space* $\mathcal{O}(n)$.

4. An algorithm using LCP

Our second algorithm for computing LPF uses the LCP array. Its advantage over the previous algorithm is that it processes the suffix array in one pass and requires less memory space. The idea is similar to the one above. Assuming we know the longest common prefixes between suf_i and either $\mathsf{suf}_{\mathsf{prev}_<[i]}$ or $\mathsf{suf}_{\mathsf{prev}_>[i]}$, it is enough to take the maximum of the two.

Using this idea, we give a space-efficient algorithm for computing LPF. For a better understanding, it is useful to arrange the SA and LCP arrays in a graph. The vertices are labelled by the SA values and the edges by the LCP values. The vertices are arranged in the left-to-right order corresponding to their order in SA and are placed at a height corresponding to their starting position in the string. In other words, if SA[i] = j, then the vertex labelled j is plotted with abscissa i and ordinate j. An example if shown by the graph in Fig. 3(i) consisting of the solid edges only.

Now consider the vertices in decreasing order of their labels, that is, vertices that are highest in the graph ("peaks") are considered first. For vertex 13, the two adjacent edges are labelled 0 and 1, corresponding to the longest common prefixes of suf_{13} with $\mathsf{suf}_{\mathsf{prev}_<[i]} = \mathsf{suf}_4$ and $\mathsf{suf}_{\mathsf{prev}_>[i]} = \mathsf{suf}_7$, respectively. Therefore, the maximum of the two gives $\mathsf{LPF}[13] = 1$. On the other hand, the minimum of the two gives the longest common prefix of suf_4 and suf_7; we remove the vertex 13 and add a (dotted) edge between 4 and 7, labelled 0. Note that $\mathsf{prev}_<[7] = 4$ and $\mathsf{prev}_>[7] = 2$, so this property is an invariant of the graph. Next we consider the vertex 12 and so on. Fig. 3(ii) shows the graph after having considered the vertices 13, 12, and 11.

It is clear that we need not consider the vertices in this order. For instance, we can compute LPF[6] right away. Precisely, any vertex which is a "peak" in our graph can have its LPF value computed. In the algorithm in Fig. 4 we consider the vertices in the order they appear in the SA (that is, left to right in the graph) and use a stack to store unprocessed vertices. The vertices in the stack will be in increasing order from bottom to top. Whenever a vertex with a smaller label than the one on top of the stack is encountered, the top of the stack is processed and popped. For instance, vertex 4 has to wait for 13 and 7 to be processed. In order to process all vertices uniformly, we assume a last (virtual) vertex with label n and height -1. The stack contains pairs

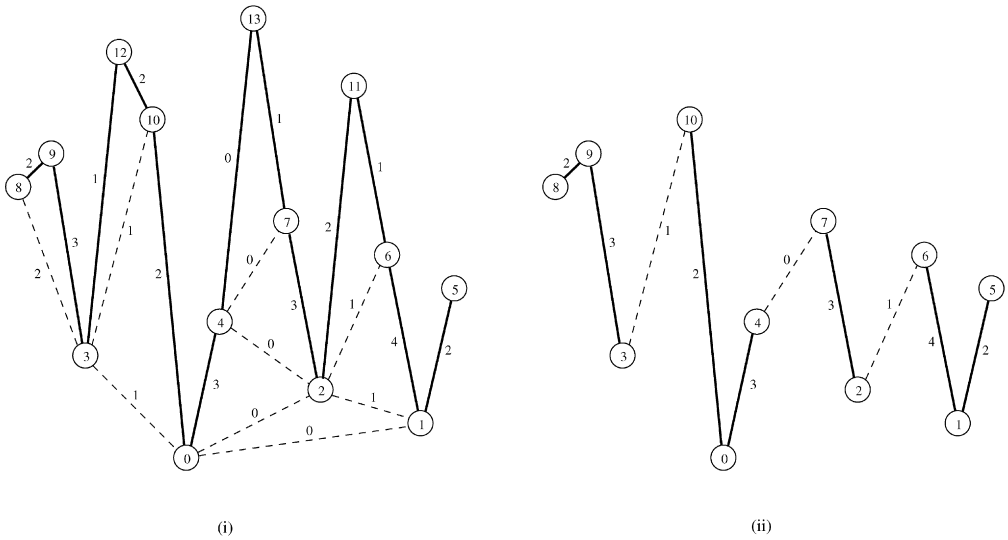

Fig. 3. (i) Solid edges form the graph representing SA and LCP for the text abbaabbbaaabab; dotted edges show the (conceptual) transformation of the graph during the algorithm COMPUTE_LPF_USING_LCP. (ii) The graph after the vertices 13, 12, and 11 were considered.

```
COMPUTE_LPF_USING_LCP(SA, LCP)
    SA[n] ← −1; LCP[n] ← 0
    PUSH((0, SA[0]), 𝒮)
    for i from 1 to n do
      lcp ← LCP[i]
      while ((𝒮 ≠ ∅) and (SA[i] < TOP(𝒮).pos))
        LPF[TOP(𝒮).pos] ← max(TOP(𝒮).len, lcp)
        lcp ← min(TOP(𝒮).len, lcp)
        v ← TOP(𝒮)
        POP(𝒮)
        if (LPF[v.pos] = 0) then
        PrevOcc[v.pos] ← −1
        else if (v.len > lcp) then
        PrevOcc[v.pos] ← TOP(𝒮).pos
        else PrevOcc[v.pos] ← SA[i]
      if (i < n) then PUSH((lcp, SA[i]), 𝒮)
    return LPF
```

Fig. 4. Algorithm for computing LPF using LCP.

of the form $(x.\text{len}, x.\text{pos})$, where $x.\text{pos}$ is a position (in SA) and $x.\text{len}$ stores the longest common prefix between $\text{suf}_{x.\text{pos}}$ and the suffix corresponding to the node right below x in the stack (or 0 if none). For instance, when the vertex 7 is considered, 13 is processed and removed from the stack. The top of the stack becomes 4 and 7 is then pushed on top of it: $\text{TOP}(\mathcal{S}).\text{len} \leftarrow 0$ and $\text{TOP}(\mathcal{S}).\text{pos} \leftarrow 7$.

The correctness of the algorithm follows from the above discussion. It runs in $\mathcal{O}(n)$ time because each element of SA is pushed only once on to the stack. Also, [10] gives a very simple linear time algorithm to compute the LCP array. The spaced used by the stack is at most n pairs of integers, which is reached for the string $a^{n-1}b$. However, the expected size of the stack is much less.

There is an interesting consequence of the above discussion, namely that the array LPF is a permutation of LCP. This is shown by analyzing the graph in Fig. 3(i). The labels of the solid edges form the LCP array. When removing a "peak" from the graph, the maximum of the two labels of the adjacent edges becomes an LPF value whereas the minimum becomes the label of the newly formed (dotted) edge. Each value will eventually become an LPF value which proves the statement.

Proposition 1. LPF *is a permutation of* LCP.

Remark 1. Note that [1] suggests a bottom-up computation of the LPF array on the lcp-interval tree (isomorphic with the suffix tree) and therefore leads to a similar result with ours. However, our approach is much simpler.

5. Application 1: Computing the Lempel–Ziv factorization

The *Lempel–Ziv factorization* of w [14] is the decomposition $w = u_0 u_1 \ldots u_k$, where each u_i (except possibly u_k) is the longest prefix of $u_i u_{i+1} \ldots u_k$ that has another occurrence to the left in w or a single letter in case this prefix is empty. For our example the Lempel–Ziv factorization is a.b.b.a.abb.baa.ab.ab.

The Lempel–Ziv factorization is a basic and powerful technique for text compression [17]. It has many

```
LEMPEL–ZIV_FACTORIZATION(LPF)
1.  LZ[0] ← 0; i ← 0
2.  while (LZ[i] < n) do
3.     LZ[i + 1] ← LZ[i] + max(1, LPF[LZ[i]])
4.     i ← i + 1
5.  return LZ
```

Fig. 5. Algorithm for computing Lempel–Ziv factorization using LPF.

variants used in gzip or PKzip software, and more generally in dictionary compression methods.

The Lempel–Ziv factorization is easily computed from LPF. The algorithm is shown in Fig. 5. For the example text abbaabbbaaabab in Fig. 1, the algorithm outputs $\text{LZ} = [0, 1, 2, 3, 4, 7, 10, 12]$.

Theorem 2. *The Lempel–Ziv factorization of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time.*

The experimental results of [2] may be explained by the fact that suffix-tree-based algorithms computing the Lempel–Ziv factorization usually have complexity $\mathcal{O}(\log(|A|)n)$, where $|A|$ is the cardinality of the alphabet.

Also, note that suffix trees allow online computation of the Lempel–Ziv factorization. However, this comes with the extra $\log(|A|)$ factor. It remains open whether true linear-time online computation is possible.

6. Application 2: Computing runs in linear time

Repetitions are a fundamental topic in stringology and appear in many applications such as text algorithms, data compression, or analysis of biological sequences. The simplest repetition is a square ww, where w is any string. A general repetition has the form w^e, for any rational exponent $e \geqslant 2$ such that $e|w|$ is an integer; e.g., $(\text{aabab})^{7/5} = \text{aababaa}$. Particularly important turned out to be maximal repetitions [15] or *runs*. A run is an occurrence of a repetition that cannot be extended. As an example, the string aababaabba contains the runs aa at positions 0 and 5, ababa, and bb. Runs allow the encoding of all repetitions in linear space [13].

An element of the Lempel–Ziv factorization carries information already processed by any online algorithms computing repetitions. Therefore it is not surprising that the Lempel–Ziv factorization plays a central part in the algorithm of [13] as well as in all efficient computations of repetitions in strings. Their running time is then $\mathcal{O}(n \log(|A|))$. Using a suffix array and the algorithms described in previous sections leads to linear-time computations on integer alphabets. This applies to: testing square freeness of a string [3], computing all leftmost maximal periodicities [15], computing all runs [13], computing of all local periods of a string [6], and computing all primitively-rooted squares occurring in a string [8]; see also [4]. In particular, we get:

Theorem 3. *The runs of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time.*

Acknowledgements

We warmly thank Bill Smyth and Simon Puglisi for interesting discussions related to the present subject during the "String Masters" meeting held in Hamilton in July 2007.

Also, we would like to thank the anonymous referees for comments that helped improving the presentation.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlenbusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms 2 (2004) 53–86.

[2] G. Chen, S.J. Puglisi, W.F. Smyth, Fast and practical algorithms for computing all runs in a string, in: Proc. of CPM'07, in: Lecture Notes in Comput. Sci., vol. 4580, Springer, Berlin, July 2007, pp. 307–315.

[3] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 45 (1) (1986) 63–86.

[4] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge Univ. Press, 2007.

[5] M. Crochemore, L. Ilie, Computing local periodicities in strings, invited talk, in: AutoMathA'07, Palermo, Italy, June 2007.

[6] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, A. Lefebvre, Linear-time computation of local periods, Theoret. Comput. Sci. 326 (1–3) (2004) 229–240.

[7] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. of FOCS'97, IEEE Computer Society Press, 1997, pp. 137–143.

[8] D. Gusfield, J. Stoye, Linear time algorithm for finding and representing all the tandem repeats in a string, J. Comput. System Sci. 69 (2004) 525–546.

[9] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. of ICALP'03, in: Lecture Notes in Comput. Sci., vol. 2719, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 943–955.

[10] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. of CPM'01, in: Lecture Notes in Comput. Sci., vol. 2089, Springer-Verlag, Berlin, 2001, pp. 181–192.

[11] D.K. Kim, J.S. Sim, H. Park, K. Park, Constructing suffix arrays in linear time, J. Discrete Algorithms 3 (2–4) (2005) 126–142.

[12] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.

[13] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: Proc. of FOCS'99, IEEE Computer Society Press, 1999, pp. 596–604.

[14] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inform. Theory 92 (1) (1976) 75–81.
[15] M.G. Main, Detecting leftmost maximal periodicities, Discrete Appl. Math. 25 (1989) 145–153.
[16] U. Manber, G. Myers, Suffix arrays: a new method for on-line search, SIAM J. Comput. 22 (5) (1993) 935–948.
[17] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.