Efficient Computing of Longest Previous Reverse Factors

Supaporn Chairungsee
King's College London
London WC2R 2LS, UK
e-mail: supaporn.chairungsee@kcl.ac.uk

Maxime Crochemore
King's College London
London WC2R 2LS, UK
e-mail: maxime.crochemore@kcl.ac.uk

ABSTRACT

We study the problem of finding the longest previous reverse factor occurring at each position of a string. This is a generalisation of a notion used for the optimal detection of various types of palindromes in a string. We describe two algorithms for computing the LPrF table of a string, one from its Suffix Tree and the second from its Suffix Automaton respectively. These algorithms run in linear time on a fixed size alphabet. Typical applications of these algorithms are for RNA secondary structure prediction and for text compression.

Keywords

longest previous reverse factor, palindrome, Suffix Tree, Suffix Automaton, RNA secondary structure prediction, text compression

1. INTRODUCTION

The problem is to compute efficiently, for a given string y, the LPrF table that stores at each index i the maximal length of factors (subwords) that both start at position i on y and occur reversed at a smaller position. Here is the example table for the string $y = aababaabab$:

position i	0	1	2	3	4	5	6	7	8	9
$y[i]$	a	a	b	a	b	a	a	b	a	b
LPrF$[i]$	0	1	0	1	3	2	4	3	2	1

The LPrF table is a concept close to the LPF table for which the previous occurrence in not reversed (see [4]). This latter table extends the Ziv-Lempel factorisation of a text [10] intensively used for conservative text compression (see [1]).

The LPrF table generalises a factorisation of strings used by Kolpakov and Kucherov [8] to extract certain types of palindromes in molecular sequences. Their palindromes are of the form uvw where v is a short string and w is the complemented reversed string associated with u (complement consists in exchanging letters A and U, as well as C and G, the Watson-Crick pairs of nucleotides). These palindromes play an important role in RNA secondary structure prediction because they signal potential hair-pin loops in RNA folding (see [2]). In addition the reverse complement of a factor has to be considered up to some degree of approximation.

An additional motivation for considering the LPrF table is text compression. Indeed, it may be used, in connection with the LPF table, to improve the Ziv-Lempel factorisation (basis of several popular compression software, see for example [9]) by considering reverse factors as well as usual factors occurrences. The feature has already been implemented in [7] but without LPrF and LPF tables, and our algorithms provide more time-efficient techniques to compress DNA sequences under the scheme.

As far as we know, the LPrF table of a string has never been considered before. Our source of inspiration was the notion of LPF table and the optimal methods for computing it in [4]. It is shown there that the LPF table can be derived from the Suffix Array of the input string both in linear time and with only a constant amount of additional extra space. The Suffix Array of a string is a data structure storing both the list (of positions) of its suffixes in the lexicographic order and the maximal length of common prefixes between consecutive suffixes in the list (see [3]).

In this note, we present algorithms for computing efficiently the LPrF table of a string from its Suffix Tree and from its Suffix Automaton, respectively. The algorithms run in linear time, that is, $O(n)$ time for a string of length n, provided the alphabet is of fixed size. On a general alphabet the running time becomes $O(n \log a)$ where a is the number of different letters occurring in the input string. Note that a straightforward algorithm would run in cubic time, or in quadratic time if a linear-time string-matching algorithm is used. So, we get the same running time as the algorithm described in [8] for the corresponding factorisation although our algorithm produces more information stored in the table and ready to use. Based on it, the factorisations of strings used for text compression and for designing string algorithms may be further optimised. Our algorithms rely on a clever usage of the notion of suffix links of the data structures, links that are essential for their linear-time construction but which the factorisation algorithm in [8] does not take advantage of.

It has been shown recently in [5] that using the Suffix Array of the input text the computation of the LPrF table can be done in linear-time on an integer alphabet. The result relaxes slightly the condition on the alphabet used in the solutions described below but the algorithm utilises an additional data structure for answering in constant time Range Minimum Queries (see for example [6]), which makes the overall process more complex.

The question of whether a linear-time algorithm exists under the condition of the string being drawn from a general alphabet seems to be out of reach.

After introducing the notation in the next section, we describe the LPrF table computation with a Suffix Tree in Section 3 and with a Suffix Automaton in Section 4. A conclusion follows.

2. BASIC DEFINITIONS

In this section, we briefly recall the notions of Suffix Trie, Suffix Tree, Suffix Automaton and then formally define the LPrF table.

The Suffix Trie of a string is the deterministic automaton that recognises the set of suffixes of the string and in which two different paths of same source always have distinct ends [3]. Thus, the graph structure of the automaton is a tree whose arcs are labelled by letters. The Suffix Trie of the string w is denoted by $\mathcal{T}(w)$. Its nodes are the factors of w, the empty string is the initial state (the root), and the suffixes of w are the terminal states. We define $s\ell[q]$ as the suffix link of (nonempty) state q. If $q = au$ for some letter a, then $s\ell[q] = u$.

Let $\mathcal{T}_C(w)$ denote the Suffix Tree of w. It is the compacted version of the Suffix Trie, obtained by deleting the nodes of degree 1 that are not terminal in the Suffix Trie. The labels of arcs then become strings of variable positive length and labels of two edges outgoing the same state start with two different letters. As above and with the same notation we denote by $s\ell$ the suffix link of $\mathcal{T}_C(w)$.

Let $\mathcal{S}(w)$ denote the Suffix Automaton of w. It is the minimal automaton that accepts the set of suffixes of w. We define F as the failure link of $\mathcal{S}(w)$ and consider the length function L, informally defined as follows. If state q is associated with the nonempty string u, $F[q]$ is the state associated with the longest suffix of u leading from the initial state to a state different from q. And $L[q]$ denotes maximal length of labels of path from the initial state to q.

On the string y of positive length n, the LPrF table is defined, for i, $0 \leq i < n$, by: LPrF$[i]$ is the maximal length of prefixes of $y[i \mathinner{.\,.} n-1]$ for which the reverse appears in $y[0 \mathinner{.\,.} i-1]$. The prefix is called the Longest Previous reverse Factor at position i on y. On the example table above, LPrF$[4] = 3$ because the factor baa of length 3 occurs at position 4, its reverse aab occurs at position 0 in $y[0 \mathinner{.\,.} 3]$, and no longer factor satisfies the condition.

3. USING A SUFFIX TREE TO COMPUTE LPRF

In this section we show how the LPrF table can be computed in linear time with the Suffix Tree of the input string. First, we describe the algorithm using the basic suffix data structure that is the Suffix Trie of the reverse input. Then, we show how to modify it to have it run with the Suffix Tree.

The algorithm to compute the LPrF table of y is first described with the Suffix Trie of the reverse string y^{R}, $\mathcal{T}(y^{\mathrm{R}})$. Figure 1 displays the Suffix Trie of babaababaa used to compute the LPrF table of the example string aababaabab. Not shown on the picture, the structure includes the suffix link recalled above as well as an addition attribute on state mentioned below.

The code of the algorithm below uses $\mathcal{T}(y^{\mathrm{R}})$, the Suffix Trie of the string y^{R}. Its initial state is *root* and its transition function denoted by δ. It works as follows. At a given step, i is a position on y, ℓ is the length of the current match, and q is the current state of the trie. We have the invariant $\delta(root, y[i-\ell \mathinner{.\,.} i-1]) = q$ where δ denotes the transition function of the trie. The condition to extend the match by the letter $a = y[i]$ is that both $\delta(q, a)$ is defined and the factor $y[i-\ell \mathinner{.\,.} i-1]a$ occurs in y^{R} at a position at least as large as $n-i+\ell$. This is tested on the largest position

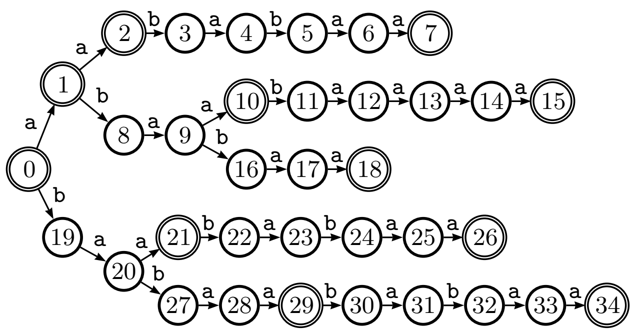

Figure 1. The Suffix Trie of babaababaa

associated with the state $\delta(q, a)$ and called $MaxPos[\delta(q, a)]$. We assume that this array is pre-computed during a linear-time traversal of the trie. If the condition is not met, the first letter of the match is implicitly deleted when using the suffix link of the trie.

The drawback of using the Suffix Trie of y^{R} is that the structure may require quadratic memory space. This is why the solution presented in this section uses the Suffix Tree of y^{R}, $\mathcal{T}_C(y^{\mathrm{R}})$, instead (see example in Figure 2). This structure can be implemented in actual linear space.

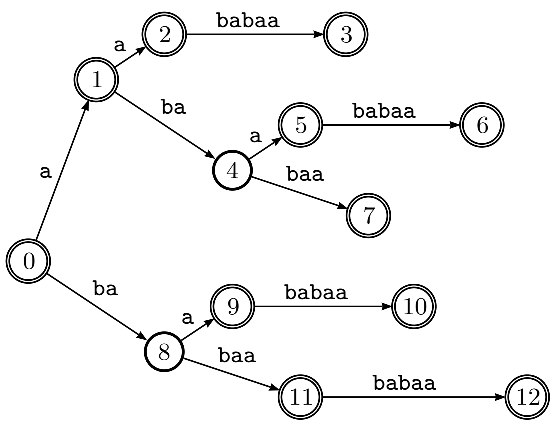

Figure 2. The Suffix Tree of babaababaa

The attributes of states ($s\ell$ and $MaxPos$) of the tree in Figure 2 are given in the following table:

state q	0	1	2	3	4	5	6	7	8	9	10	11	12
$s\ell[q]$	-	0	1	7	8	9	10	11	1	2	7	5	6
$MaxPos[q]$	10	9	8	3	6	6	1	4	7	7	2	5	0

When using a Suffix Tree instead of a Suffix Trie, the essential difference is in the implementation of the transition function and of the suffix link. Indeed, in $\mathcal{T}_C(y^{\mathrm{R}})$, arcs are of the form $(q, (p, k), r)$ corresponding to the transition $\delta(q, y[p \mathinner{.\,.} p+k-1]) = r$. We can then represent states of the trie by $(q, (p, k, k'))$ where $0 \leq k' < k$. If $k' = 0$, it is actually state q itself. Otherwise, we call it a virtual state and it corresponds to a state of the trie. Implementing a transition or a suffix link then becomes a simple exercise. Note that there is no need to compute a suffix link with a call to the specific procedure used to build the tree and called FAST-FIND in [3].

```
LPrF-TREE$(y, n)$
 1  $(q, \ell) \leftarrow (root, 0)$
 2  $i \leftarrow 0$
 3  repeat $a \leftarrow y[i]$
 4          while $(i < n)$ and $(\delta(q, a) \neq \text{NULL})$
              and $((n - i + \ell) \leq MaxPos[\delta(q, a)])$ do
 5              $(q, \ell) \leftarrow (\delta(q, a), \ell + 1)$
 6              $i \leftarrow i + 1$
 7              $a \leftarrow y[i]$
 8          LPrF$[i - \ell] \leftarrow \ell$
 9          if $q \neq root$ then
10              $(q, \ell) \leftarrow (s\ell[q], \ell - 1)$
11          else    $i \leftarrow i + 1$
12  until $(i = n)$ and $(\ell = 0)$
13  return LPrF
```

Theorem 1. *The algorithm* LPrF*-tree computes the* LPrF *table of a string of length n in time $O(n)$ on a fixed size alphabet.*

Proof. Although not obvious the correctness of the algorithm comes from the discussion above. We evaluate its running time. The preprocessing comprises the Suffix Tree construction and the computation of the *MaxPos* attribute of its nodes, which are done in linear time by known techniques. After such a preprocessing all instructions, the running time depends mainly on the number of tests done in line 4 and the number of operations to compute a suffix link in line 10. Since each test either leads to increment i or to increment $i - \ell$ the position of the current match, whose values never decrease, there is a linear number of tests. When a suffix link is computed for a virtual state $(q, (p, k, k'))$, the number of letter comparisons is no more than k'. But since these comparisons have already been done during a previous test in line 4, the total number of comparisons of letter done in line 10 is also linear. Which implies the result. ■

Note that if the alphabet is not fixed, the computation of a transition from a real state requires $O(\log a)$ time, where a is the size of the alphabet of y, to keep the memory space linear. This leads to an algorithm which overall running time is $O(n \log a)$.

4. USING A SUFFIX AUTOMATON TO COMPUTE LPRF

In this section we present another solution to compute the LPrF table of the input string y in linear time. It is done with the Suffix Automaton $\mathcal{S}(y^{\mathrm{R}})$. The structure includes the failure link F and the table L recalled above as well as an additional attribute on states SC mentioned below.

Figure 3 displays the Suffix Automaton of `babaababaa` used for computing the LPrF table of the string `aababaabab`.

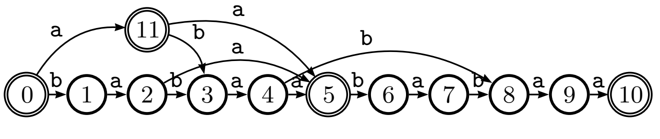

Figure 3. Suffix Automaton of `babaababaa`

The next table gives the attributes (F, L and SC) of the states of the automaton displayed in Figure 3:

state q	0	1	2	3	4	5	6	7	8	9	10	11
$F[q]$	0	0	11	1	2	11	3	4	3	4	5	0
$L[q]$	0	1	2	3	4	5	6	7	8	9	10	1
$SC[q]$	0	2	1	2	1	0	4	3	2	1	0	0

As above, at a given step i is a position on y, ℓ is the length of the current match, and q is the current state of the automaton. The principal invariant is $\delta(initial, y[i - \ell \,..\, i - 1]) = q$ where δ denotes the transition function of the automaton and *initial* its initial state. The condition to extend the match by the letter $a = y[i]$ is that $\delta(q, a)$ is defined and that $y[i - \ell \,..\, i - 1]a$ occurs in y^{R} at a position at least as large as $n - i + \ell$, as before. This can be tested efficiently on the automaton if the table SC is available. For a state r, $SC[r]$ is the minimal length of labels of paths from r to a terminal state. The table can be pre-processed via a mere traversal of the automaton (see [3]). The test becomes $i - \ell \leq \ell + 1 + SC[\delta(q, a)]$ where the first member is the length of $y[0 \,..\, i - \ell - 1]$ and the second member the minimal length of suffixes of y^{R} starting with the next match.

When the test is negative, the failure link F is applied to shorten the match whose length is given by L. None of the suffixes of the match of length larger than $L[F[q]]$, which all correspond to the same state q, is able to change the value of the test in line 4. Then, a batch of LPrF values are computed in lines 8–10.

In the code below we assume that $F[initial] = initial$. The value of $F[initial]$ is usually left undefined for Suffix Automata but the assumption simplifies the presentation of the algorithm.

```
LPrF-AUTOMATON$(y, n)$
 1  $(q, \ell) \leftarrow (initial, 0)$
 2  $i \leftarrow 0$
 3  repeat $a \leftarrow y[i]$
 4          while $(i < n)$ and $(\delta(q, a) \neq \text{NULL})$
              and $((i - \ell) \geq \ell + 1 + SC[\delta(q, a)])$ do
 5              $(q, \ell) \leftarrow (\delta(q, a), \ell + 1)$
 6              $i \leftarrow i + 1$
 7              $a \leftarrow y[i]$
 8          repeat LPrF$[i - \ell] \leftarrow \ell$
 9              $\ell \leftarrow \max\{0, \ell - 1\}$
10          until $\ell = L[F[q]]$
11          if $q \neq initial$ then
12              $q \leftarrow F[q]$
13          else    $i \leftarrow i + 1$
14  until $(i = n)$ and $(\ell = 0)$
15  return LPrF
```

Theorem 2. *The algorithm* LPrF-AUTOMATON *computes the* LPrF *table of a string of length n in time $O(n)$ on a fixed size alphabet.*

Proof. We evaluate the running time of the algorithm. The preprocessing consists of the automaton construction with table F and L, and of the computation of the attribute SC of its states. All the preprocessing can be done in linear time (see [3]).
The running time of the rest depends on the number of tests in line 4 and in line 10. The former either leads to increment i or to execute the next instruction. The latter yields an increment of the expression $i - \ell$. Since the values of these two expressions never decrease, only n of them are executed, which implies the result. ■

As with the Suffix Tree solution, if the automaton is implemented in linear space, the overall algorithm runs in $O(n \log a)$ time.

Note that the automaton can be built in linear-time if it is implemented in space $O(an)$ with the technique of sparse matrix implementation, which would provide a linear-time solution to the detriment of the memory usage, which is usually unsatisfactory for applications.

5. CONCLUSION

We have shown that the LPrF table of a string, generalising the technique used in [8] to locate special types of biological palindromes, can be computed in linear time on a fixed size alphabet and in $O(n \log a)$ time otherwise in a linear memory space.

There is still an interesting open question about the problem: does there exists a straight computation of the table using none of the suffix data structures and running in linear time in the comparison model? A positive answer to this question is likely to introduce an exciting novel technique in Stringology.

REFERENCES

[1] T. C. Bell, J. G. Clearly, and I. H. Witten. *Text Compression*. Prentice Hall Inc., New Jersey, 1990.

[2] H.-J. Böckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics*. Springer, Berlin, 2007.

[3] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, UK, 2007.

[4] M. Crochemore, L. Ilie, C. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. LPF computation revisited. In J. Kratochvil and M. Miller, editors, *International Workshop on Combinatorial Algorithms*, LNCS, Berlin, 2009. Springer. To appear.

[5] M. Crochemore, C. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. Efficient algorithms for two extensions of the LPF table: the power of Suffix Arrays. 2009. Submitted.

[6] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5–7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[7] S. Grumbach and F. Tahi. Compression of dna sequences. In *Data Compression Conference*, pages 340–350, 1993.

[8] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. In P. Ferragina and G. M. Landau, editors, *Combinatorial Pattern Matching, 19th Annual Symposium, Pisa, Italy, June 18-20, 2008*, volume 5029 of *Lecture Notes in Computer Science*, pages 18–30, Berlin, 2008. Springer.

[9] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.

[10] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pages 337–343, 1977.