Efficient Algorithms for Longest Closed Factor Array

Hideo Bannai[1], Shunsuke Inenaga[1], Tomasz Kociumaka[2,⋆], Arnaud Lefebvre[3], Jakub Radoszewski[2,⋆⋆,⋆⋆⋆], Wojciech Rytter[2,†], Shiho Sugimoto[1], and Tomasz Waleń[2,⋆⋆]

[1] Department of Informatics,
Graduate School of Information Science and Electrical Engineering,
Kyushu University, Japan
`[bannai,inenaga,shiho.sugimoto]@inf.kyushu-u.ac.jp`
[2] Faculty of Mathematics, Informatics and Mechanics,
University of Warsaw, Warsaw, Poland
`[kociumaka,jrad,rytter,walen]@mimuw.edu.pl`
[3] Normandie Université, LITIS EA4108, NormaStic CNRS FR 3638, IRIB,
Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France
`arnaud.lefebvre@univ-rouen.fr`

Abstract. We consider a family of strings called closed strings and a related array of Longest Closed Factors (LCF). We show that the reconstruction of a string from its LCF array is easier than the construction and verification of this array. Moreover, the reconstructed string is unique. We improve also the time of construction/verification, reducing it from $\mathcal{O}(n \log n/ \log\log n)$ (the best previously known) to $\mathcal{O}(n\sqrt{\log n})$. We use connections between the LCF array and the longest previous/next factor arrays.

1 Introduction

A *closed string* is a string with a proper (possibly empty) factor that occurs in the string as a prefix and as a suffix, but not elsewhere. For example, a, abaab, and ababababa are closed strings (with the corresponding factors ε, ab, and abababa), whereas abaca and abc are not. Closed strings were first defined by Fici in [8] and since then have found applications, mostly in the field of combinatorics on words. Closed prefixes of Sturmian words were studied in [10]. A relation between closed factors and palindromic factors of a string was studied in [3]. The first algorithmic study of closed factors and closed factorizations was presented in [2].

⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

⋆⋆ Supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no 0392/IP3/2015/73.

⋆⋆⋆ Receives financial support of Foundation for Polish Science.

† Supported by the Polish National Science Center, grant no 2014/13/B/ST6/00770.

The *longest closed factor array* (LCF array) of a string X stores for every suffix of X the length of its longest closed prefix. It was introduced in [2] in connection with closed factorizations of a string. In [2] an $\mathcal{O}(n \log n / \log \log n)$-time algorithm for computing this array for a string of length n was presented. Here we consider the problem of reconstructing a string from its LCF array.

We show that a (correct) LCF array corresponds to exactly one string, up to a permutation of the alphabet. We present an $\mathcal{O}(n)$-time randomized and an $\mathcal{O}(n \min(\log |\Sigma|, \log \log n + \frac{\log^2 \log |\Sigma|}{\log \log \log |\Sigma|}))$-time deterministic algorithm for reconstructing such a string if it exists. Here Σ is the alphabet of the string. Finally we present an $\mathcal{O}(n\sqrt{\log n})$-time construction algorithm for LCF array which improves the algorithm of [2]. We use it for verification of the LCF array, that is, for checking if it corresponds to any string. As a by-product we obtain $\mathcal{O}(n\sqrt{\log n})$-time computation of the so-called closest (rightmost) Longest Previous Factor array. The complexity of the LCF construction algorithm depends on the assumption of a linearly sortable alphabet of the input string.

2 Preliminaries

Let X be a string of length n composed of characters $X[1], \ldots, X[n]$. We denote $|X| = n$. By $X[i..j]$ we denote a factor of X consisting of the letters $X[i], \ldots, X[j]$. A factor is called a prefix (suffix) of X if $i = 1$ ($j = n$ respectively). A factor is called proper if $i > 1$ or $j < n$. If $i > j$ then the factor is assumed to be the empty string ε. A border of X is a factor of X that occurs as a prefix and as a suffix of X. By $border(X)$ we denote the length of the longest proper border of X.

The string X is called *closed* if it has a proper border that does not occur elsewhere in X. In particular, every single-letter string is closed. It is easy to see that a string is closed if and only if its longest proper border does not occur elsewhere in X. The *longest closed factor array* of X is an array $LCF[1..n]$ such that $LCF[i]$ is the length of the longest prefix of $X[i..n]$ that is closed. We denote by $lcf[i]$ the factor $X[i..i + LCF[i] - 1]$.

The *longest next factor array* of X is an array $LNF[1..n]$ such that $LNF[i]$ is the length of the longest prefix of $X[i..n]$ that is a factor of $X[i + 1..n]$. We denote by $lnf[i]$ the factor $X[i..i + LNF[i] - 1]$.

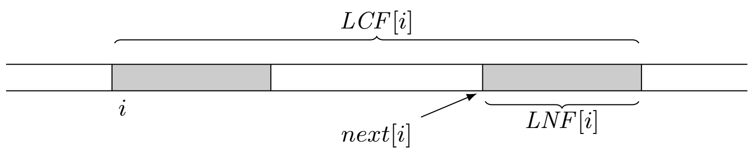

Fig. 1. Illustration of LCF, LNF and $next$ arrays. Here $LCF[i] = |lcf[i]|$ is the length of the longest closed factor starting at position i and $LNF[i] = |lnf[i]|$ is the length of the longest factor starting at position i that occurs to the right of position i.

Example 1. $X =$ `abaababababbabbb` has the following *LCF* and *LNF* arrays:

position i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$X[i]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b
$LCF[i]$	6	5	2	6	5	4	7	6	5	3	1	3	2	1
$LNF[i]$	3	2	1	4	3	2	4	3	2	1	0	2	1	0

Here the $lcf[\,]$ array is as follows: [`abaaba`, `baaba`, `aa`, `ababab`, `babab`, `abab`, `babbabb`, `abbabb`, `bbabb`, `bab`, `a`, `bbb`, `bb`, `b`].

3 Reconstruction Algorithm

In this section we show efficient reconstruction of a string from its *LCF* array. It is based on uniqueness of the output.

3.1 Uniqueness of Reconstruction

The following fact shows a correspondence between longest closed factors and longest next factors. We denote $next[i] = i + LCF[i] - LNF[i]$; see Fig. 1.

Lemma 2. *The longest proper border of* $lcf[i]$ *is* $lnf[i]$.

Proof. Let $\ell = border(lcf[i])$ and $t = i + LCF[i] - \ell$ $(t > i)$. $X[i..i + \ell - 1]$ occurs also at the position t in X. Therefore, by the definition of the longest next factor, $\ell \leq LNF[i]$. We will show that $\ell = LNF[i]$. Assume to the contrary that $\ell < LNF[i]$. Let j be the first position greater than i where $lnf[i]$ occurs in X. Consider the factor $Y = X[i..j + LNF[i] - 1]$.

Claim. Y is closed.

Proof. Y has a border $lnf[i]$. Moreover, it is its longest border, as any longer border would imply a next factor longer than $lnf[i]$. By the definition of j, the string $lnf[i]$ occurs exactly twice in Y. Hence, Y is a closed factor of X. □

Claim. $|Y| > LCF[i]$.

Proof. We have $|Y| = j + LNF[i] - i$ and $LCF[i] = t + \ell - i$. By the assumption, $LNF[i] > \ell$. The longest proper border of $lcf[i]$ is a prefix of $lnf[i]$ and so occurs at position j. If $j < t$, then this would mean the third occurrence within $lcf[i]$, which is impossible. Hence, $j \geq t$. Consequently, $|Y| > LCF[i]$. □

We conclude that Y is a closed factor longer than $lcf[i]$, a contradiction. □

We proceed with the first algorithm for recovering a string X from its *LCF* array. The algorithm is simple for positions i where $LCF[i] = 1$.

Fact 3. *The number of distinct letters of X equals the number of 1-entries in its LCF array. Moreover, each i such that $LCF[i] = 1$ corresponds to the rightmost occurrence of one of the letters in X.*

If $LCF[i] > 1$ then $X[i]$ can be recovered from one of following positions in X using the function below.

```
Algorithm ComputeSingleSymbol($i$)
Input: $X[i+1..n]$, $LCF[i..n]$
Output: $X[i]$
$bord := border(X[i+1..i+LCF[i]-1]);$
$X[i] := X[i+LCF[i]-1-bord];$
```

In the correctness proof of the function we use the following auxiliary lemma.

Lemma 4. *If $LCF[i] > 1$ then $border(X[i+1..i+LCF[i]-1]) = LNF[i]-1$.*

Proof. Let $S = X[i+1..i+LNF[i]-1]$ and $T = X[i+1..i+LCF[i]-1]$. By Lemma 2, $lnf[i]$ is a border of $lcf[i]$. Hence, S is a border of T. Assume to the contrary that T has a longer proper border S'. Then $lnf[i] = X[i]S$ is a suffix of S', as it is a suffix of T. Hence, $lnf[i]$ occurs also at position $i+1+|S'|-LNF[i] < i+LCF[i]-LNF[i] = next[i]$, which contradicts the fact that $lcf[i]$ is closed. $\square$

Lemma 5. *Assume $LCF[i] > 1$. Then the function ComputeSingleSymbol correctly computes $X[i]$.*

Proof. By Lemma 4, we have $bord = LNF[i]-1$. Consequently, $next[i] = i + LCF[i] - 1 - bord$, which yields the correctness of the function. $\square$

Fact 3 and Lemma 5 show that a string can be restored from its LCF array.

Theorem 6. *If there exists a string with the given LCF array then it is uniquely determined up to permutation of the corresponding alphabet.*

Proof. For each position i such that $LCF[i] = 1$ we introduce a unique letter $X[i]$. For each of the remaining positions, $X[i]$ can be determined from the following letters using function *ComputeSingleSymbol*. $\square$

The algorithm of Theorem 6 is not efficient yet. In the following section we introduce an additional combinatorial fact and algorithmic tools that make the solution efficient.

3.2 Efficient Reconstruction

In the reconstruction algorithm we compute the string X together with the corresponding LNF array. We use the following crucial fact.

Lemma 7. *For every $1 \le i < n$, $LNF[i] \le LNF[i+1]+1$.*

Proof. Assume to the contrary that for some i, $LNF[i] > LNF[i+1]+1$. Recall that $lnf[i]$ occurs at position $next[i]$ in X. This concludes that the string $Y = X[i+1..i+LNF[i]-1]$ of length $LNF[i]-1 > LNF[i+1]$ occurs at position $next[i]+1 > i+1$. This contradicts the definition of $LNF[i+1]$. □

In the pseudocode of algorithm *Reconstruction* we use Lemma 7 on top of the reconstruction algorithm from the previous section, based on function *ComputeSingleSymbol*. The alphabet of the reconstructed string is $\{1, 2, \ldots\}$.

```
Algorithm Reconstruction
Input: $LCF[1..n]$ array
Output: the corresponding string $X[1..n]$
$bord := 0$;
for $i := n$ downto 1 do
    { Invariant: If $i < n$ then $bord = LNF[i+1]$. }

    if $LCF[i] = 1$ then
        $X[i] := NewLetter()$;
        $bord := 0$; { $LNF[i] = 0$ }
    else

        { Efficient implementation of ComputeSingleSymbol($i$) }
        while $X[i+1..i+bord] \neq X[i+LCF[i]-bord..i+LCF[i]-1]$ do
            { $bord > border(X[i+1..i+LCF[i]-1])$ }
            $bord := bord - 1$;
        $X[i] := X[i+LCF[i]-1-bord]$;
        $bord := bord + 1$; { $LNF[i] = bord$ }

return $X$;
```

Clearly, the total number of steps of the while-loop in the algorithm *Reconstruction* is at most n. Hence, the time complexity of the algorithm depends on how fast we can check equality of two factors of a string, with the letters of the string being appended on-line from right to left.

Theorem 8. *A string of length n over alphabet Σ can be reconstructed from its longest closed factor array with an $\mathcal{O}(n)$-time randomized algorithm or an $\mathcal{O}(n \min(\log|\Sigma|, \log\log n + \frac{\log^2 \log|\Sigma|}{\log\log\log|\Sigma|}))$-time deterministic algorithm.*

Proof. For the randomized algorithm, we use Karp-Rabin fingerprinting (see e.g. [6]) to check equality of factors of the string given on-line in $\mathcal{O}(1)$ time.

For the deterministic algorithm, we use one of the incremental suffix tree constructions. The first one is the algorithm by Blumer et al. [4] which computes suffix trees for growing suffixes of the string in $\mathcal{O}(n \log|\Sigma|)$ total time. The other comes from a recent paper by Fischer and Gawrychowski [9], where the authors show how to update the suffix tree in $O(\log\log n + \frac{\log^2 \log|\Sigma|}{\log\log\log|\Sigma|})$ time after prepending a character (see Corollary 4 and Theorem 5 in [9]). Finally, given a suffix tree, determining equality of factors reduces to LCA-computation. This can be done, however, with $\mathcal{O}(1)$ overhead using LCA queries for a dynamic tree; see [5]. □

Theorem 8 provides an efficient reconstruction algorithm from the *LCF* array only if the corresponding string exists. Otherwise the reconstruction algorithm may fail or reconstruct a string which does not have the given *LCF* array. In the latter case it suffices to construct its *LCF* array and check if it matches the input *LCF* array. We deal with this case in the following section.

4 LCF Array Computation and Verification

A successor of an integer x in a set X is defined as $succ(x, X) = \min\{y \in X : y > x\}$. For an integer array $A[1..n]$, a range successor query consists in computing $succ(x, \{A[i], \ldots, A[j]\})$ for any $1 \leq i \leq j \leq n$. Babenko et al. ([1], Section 2.5) show the following result:

Lemma 9 ([1]). *A collection of q range successor queries in an array of length n can be answered offline in $\mathcal{O}((n+q)\sqrt{\log n})$ time.*

As shown in [2], computation of the LCF array reduces to $\mathcal{O}(n)$ range successor queries in the suffix array of the string. Hence, we can use Lemma 9 to improve the running time of both LCF array computation and verification.

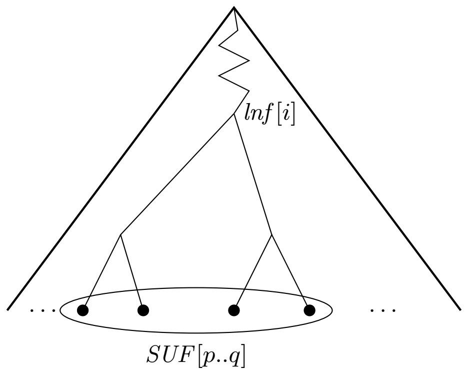

Fig. 2. Illustration of the formula $next[i] = succ(i, SUF[p..q])$. To compute $next[i]$, we need to find the first occurrence of $lnf[i]$ after the position i.

Theorem 10.
(a) The LCF array of a string can be computed in $\mathcal{O}(n\sqrt{\log n})$ time.
(b) Verification of LCF array can be done in $\mathcal{O}(n\sqrt{\log n})$ randomized time or $\mathcal{O}(n(\sqrt{\log n} + \frac{\log^2 \log |\Sigma|}{\log\log\log |\Sigma|}))$ deterministic time.

Proof. For the start, recall that a suffix tree of a string (that is, a compact trie of its suffixes) over a linearly-sortable alphabet can be constructed in $\mathcal{O}(n)$ time [6].

We reformulate the algorithm of [2]. Recall that $LCF[i] = LNF[i] + next[i] - i$. The LNF array, together with the nodes of the suffix tree corresponding to $lnf[i]$, can be computed in $\mathcal{O}(n)$ time; see Lemma 6 in [2]. Let $SUF[p..q]$ be the sequence of leaves in the suffix tree corresponding to the subtree corresponding to $lnf[i]$. Then $next[i] = succ(i, \{SUF[p], \ldots, SUF[q]\})$; see Fig. 2.

Thus Lemma 9 implies an $\mathcal{O}(n\sqrt{\log n})$-time algorithm for computing LCF array. Part (b) follows from part (a) and Theorem 8. $\square$

5 Final Remarks

For a string X of length n, the longest previous factor (LPF) array and the closest longest previous factor ($prev$) array are defined as follows. For $i = 1, \ldots, n$, $LPF[i]$ is the maximum ℓ such that $X[j..j+\ell-1] = X[i..i+\ell-1]$, for some $j < i$. For $i = 1, \ldots, n$, $prev[i]$ is the maximum $j < i$ such that $X[j..j + LPF[i] - 1] = X[i..i + LPF[i] - 1]$. Note that the LPF array is *not* the same as the LNF array of the reversed string.

The LPF array can be computed in $\mathcal{O}(n)$ time [7]. Using exactly the same approach as in Section 4 but with range predecessors instead of range successors, we obtain the following result:

Theorem 11. *The closest longest previous factor array prev of a string of length n can be computed in $\mathcal{O}(n\sqrt{\log n})$ time.*

Our Theorem 6 provides an example of a unique reconstruction of a string from its closed factors. Another example is Theorem 9 in [10], which states that every (finite or infinite) Sturmian word is uniquely determined, up to isomorphisms of the alphabet, by its sequence of open and closed prefixes. Both results are quite independent (none follows from the other).

Acknowledgements. The authors thank the participants of StringMasters workshops in 2014 in Prague and in 2015 in Warsaw, in particular Gabriele Fici, Ondřej Guth, and Jan Holub, for helpful discussions.

References

1. M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. *arXiv:1408.6182v4*, 2015.
2. G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. In J. Holub and J. Žďárek, editors, *Prague Stringology Conference 2014*, pages 162–168, 2014.
3. G. Badkobeh, G. Fici, and Z. Lipták. On the number of closed factors in a word. In A. H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications - LATA 2015*, volume 8977 of *Lecture Notes in Computer Science*, pages 381–390. Springer, 2015.

4. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.
5. R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
6. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, Cambridge, 2007.
7. M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. Computing the longest previous factor. *Eur. J. of Comb.*, 34(1):15–26, 2013.
8. G. Fici. A classification of trapezoidal words. In P. Ambroz, S. Holub, and Z. Masáková, editors, *Combinatorics on Words - WORDS 2011*, volume 63 of *EPTCS*, pages 129–137, 2011.
9. J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In F. Cicalese, E. Porat, and U. Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015*, volume 9133 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015.
10. A. D. Luca and G. Fici. Open and closed prefixes of Sturmian words. In J. Karhumäki, A. Lepistö, and L. Q. Zamboni, editors, *Combinatorics on Words - WORDS 2013*, volume 8079 of *Lecture Notes in Computer Science*, pages 132–142. Springer, 2013.