Journal of Discrete Algorithms 20 (2013) 61–64

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

A note on a simple computation of the maximal suffix of a string

Zbigniew Adamczyk[a], Wojciech Rytter[a,b,*]

[a] Copernicus University, Faculty of Mathematics and Informatics, Poland
[b] Institute of Informatics, Warsaw University, Poland

ARTICLE INFO

Article history:
Available online 21 March 2013

Keywords:
Algorithm
Constant space
Linear time
Maximal suffix

ABSTRACT

We present an alternative linear-time constant-space algorithm for computing maximal suffixes in strings, which is interesting due to its especially compact description. It also provides an exercise in proving nontrivial correctness of an algorithm having a concise description. The results are of a methodological character.

1. Introduction

Usually in algorithmics we are interested in the reduction of time/space complexity (in sequential computations), but in this note the main issue is structural complexity – simplicity of the algorithm description. Only algorithms working in $O(n)$ time and $O(1)$ space are considered here.

Maximal suffixes of strings play an important role, for example in constant-space string-matching, see [3,6,4,1], and Lyndon factorization.

Maximal suffix computation is from [3] with a complete proof. It is adapted from the Lyndon factorization in [4], which computes minimal suffixes, but slightly simpler.

Here we design an alternative algorithm using ideas related to the constant-space algorithm for equivalence of cyclic shifts, see [8,7].

Assume w is an input string of size n, where the positions are numbered from 0 to $n-1$.

Denote by $\mathit{MaxSuf}(w)$ the lexicographically maximal suffix of w, and by $\mathit{MaxSufPos}(w)$ its starting position.

Example 1.1. If $w = abaaabaaababab$ then $\mathit{MaxSuf}(w) = 9$.

We will use some combinatorial properties of strings.

Denote by *period*(x) the shortest (string) period of x, and let *per*(x) denote the length of the shortest period. A string x is *border-free* iff $per(x) = |x|$ and it is said to be *self-maximal* iff $\mathit{MaxSuf}(x) = x$.

Example 1.2. The string $x = babaabab$ is self-maximal.

Observe that $period(x) = babaa$ is border-free.

[*] Corresponding author.
E-mail address: rytter@mimuw.edu.pl (W. Rytter).

1570-8667/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2013.03.002

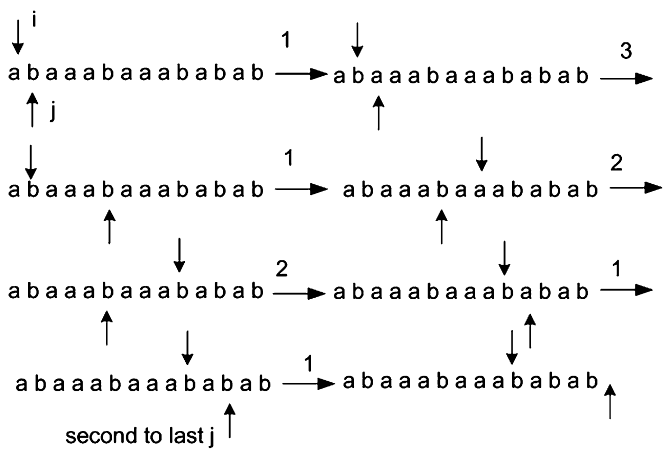

Fig. 1. An illustration of the execution of the algorithm for an example string: the numbers over arrows denote the number of iterations of the main (outer) *while-loop*. Observe that the difference between the second to last j and the value of the last i equals the period of $\mathrm{M}axSuf(w)$.

2. The algorithm

Our main result is the descriptional simplicity of the following algorithm which computes the starting position of the maximal suffix of a string.

```
ALGORITHM Compute-MaxSufPos($w$)
$i := 0$; $j := 1$;
while $j < n$ do
  $k := 0$;
  while $j + k < n - 1$ and $w[i + k] = w[j + k]$ do $k := k + 1$;
  if $w[i + k] < w[j + k]$ then $i := i + k + 1$ else $j := j + k + 1$;
  if $i = j$ then $j := j + 1$;
return $i$;
```

The algorithm obviously works in (additional) constant space and linear time (each comparison causes one of i or j to increase).

Performance of the algorithm is illustrated for an example string in Fig. 1.

3. Correctness of the algorithm

Correctness of the algorithm is nontrivial. The following well-known fact is needed.

Lemma 3.1. *(See [2,4].) The shortest string period of the maximal suffix is border-free.*

Theorem 3.2. *The algorithm correctly returns $i = \mathrm{M}axSufPos(w)$.*

Proof. Let $(p, q) \to (p', q')$ mean that from the configuration (p, q) in one iteration we go to (p', q'), and let $\to^*$ be the transitive closure of the relation $\to$.

Claim 3.3. *We have the following invariant after each main iteration, where we denote $u = w[i..j - 1]$:*

$(*)$ $(i < j < n) \Rightarrow$ *u is self-maximal and $per(u) = |u|$.*
$(**)$ *The maximal suffix of w does not start before i.*

Proof of the claim. Initially $i = 0$, $j = 1$ and the invariant holds.

Let us consider the iteration when i is moved for the first time. It is easy to see that before this iteration the invariant holds and the word $u = w[i..j-1]$ is self-maximal. The value of i has moved for the first time from $i = 0$ to $i = i' = i+k+1$, see Fig. 2.

Then $w[0..j + k] = u^t \cdot vb$, where $|v| < |u|$, $u < vb$, see Fig. 2.
Denote $m = |u^t|$, then the (partial) history of the algorithm is as follows:

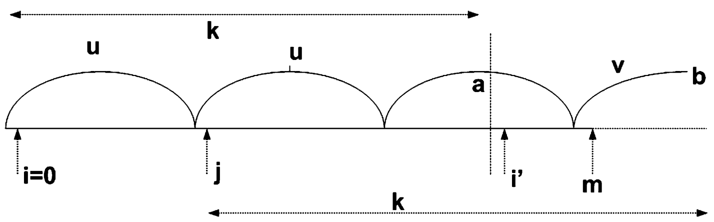

Fig. 2. The configuration when i is moved for the first time to some position i': $u^t vb$ is a prefix of w, $MaxSuf(u^t vb) = vb$, $m = |v^t|$, and the maximal suffix of the whole text does not start before m.

$$(i, j) \to (i', j) \to^* (m, j) \to^* (m, m+1).$$

The word u is the shortest period of a self-maximal word $u^t v$, and therefore Lemma 3.1 implies that u is border-free.

Consequently, whenever we start at any position i in the range $[i', m-1]$, the next position for i cannot be greater than m. Otherwise we would start with i inside an occurrence of u and go to the end of u, matching a prefix z of u, so u would have a border z, a contradiction of Lemma 3.1.

Hence the value of i will be moving from i' until it reaches m, at which point j starts to increase until reaching $m+1$.

When (i, j) becomes $(m, m+1)$ we can cut off the prefix u^k of the text, and the whole computation starts again from the beginning (m can be treated as zero). Now the claim for w follows from the claim for a shorter string. Finally, j goes beyond the scope of the text. This completes the proof of the claim. □

Proof of the thesis. Consider the last value of i and the second to last value of j.

According to the invariant $(*)$ we have: $u = w[i..j-1]$ is self-maximal. Also u is a period of $w[i..n-1]$ (the suffix of the whole text).

Observe now that (generally) if a string w' has a prefix u which is both self-maximal and a period of w', then w' is also self-maximal.

Consequently, the word $w[i..n-1]$ is self-maximal, and it is the maximal suffix of w. This completes the proof of the theorem.

Our algorithm, similarly to Duval's algorithm, see [5], can also output the shortest period of the maximal suffix. The following fact follows directly from the proof of Theorem 3.2, where $u = w[i..j-1]$ is the shortest period of the maximal suffix. □

Observation 3.4. *Assume j' is the second to last value of j in the algorithm.*
If $i = MaxSufPos(w) < n-1$ then $j' - i = per(MaxSuf(w))$.

4. Final remarks

We can try to speed up our algorithm (at the cost of descriptional complexity). When i moves to the right we can move (i, j) in one step to $(m, m+1)$ reducing potentially many iterations to one, see Fig. 2. Observe that

$$m = i + \left(\left\lfloor \frac{k}{j-i} \right\rfloor + 1\right) \cdot (j-i). \quad (1)$$

Hence a faster algorithm can be obtained using Eq. (1). The statement $i := i + k + 1$ is to be replaced by

$$i := i + \left(\left\lfloor \frac{k}{j-i} \right\rfloor + 1\right) \cdot (j-i); \quad j := i + 1.$$

The faster algorithm is shown below (for completeness).

```
$i := 0$; $j := 1$;
**while** $j < n$ **do**
  $k := 0$;
  **while** $j + k < n - 1$ and $w[i+k] = w[j+k]$ **do** $k := k + 1$;
  **if** $w[i+k] < w[j+k]$ **then**
    $i := i + (\lfloor \frac{k}{j-i} \rfloor + 1) \cdot (j - i)$; $j := i + 1$
  **else** $j := j + k + 1$;
**return** $i$;
```

Such an algorithm becomes a disguised version of Duval's algorithm. Conversely, we could say that algorithm Compute-M*axSufPos* is a disguised and slightly slowed-down (but still working in linear time) version of Duval's algorithm, yet having simpler description.

The faster version of algorithm Compute-M*axSufPos* loses its simplicity because of integer division and multiplication, due to Eq. (1). These operations could be eliminated by using only addition and subtraction but this would decrease simplicity even more.

However, simplicity of the description was our main issue, and from this point of view algorithm Compute-M*axSufPos* is much better.

References

[1] Maxime Crochemore, String-matching on ordered alphabets, Theor. Comput. Sci. 92 (1) (1992) 33–47.
[2] Maxime Crochemore, Christophe Hancart, Thierry Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[3] Maxime Crochemore, Dominique Perrin, Two-way string matching, J. ACM 38 (3) (1991) 651–675.
[4] Maxime Crochemore, Wojciech Rytter, Text Algorithms, Oxford Press, 1994.
[5] Jean-Pierre Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (4) (1983) 363–381.
[6] Wojciech Rytter, On maximal suffixes, constant-space linear-time versions of KMP algorithm, Theor. Comput. Sci. 299 (1–3) (2003) 763–774.
[7] Yossi Shiloach, A fast equivalence-checking algorithm for circular lists, Inf. Process. Lett. 8 (5) (1979) 236–238.
[8] Yossi Shiloach, Fast canonization of circular strings, J. Algorithms 2 (2) (1981) 107–121.