Ghuman, Sukhpal Singh; Giaquinta, Emanuele; Tarhio, Jorma

Lyndon Factorization Algorithms for Small Alphabets and Run-Length Encoded Strings

Published in:
Algorithms

DOI:
10.3390/a12060124

Published: 01/06/2019

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY

Please cite the original version:
Ghuman, S. S., Giaquinta, E., & Tarhio, J. (2019). Lyndon Factorization Algorithms for Small Alphabets and Run-Length Encoded Strings. *Algorithms*, *12*(6), Article 124. https://doi.org/10.3390/a12060124

algorithms

MDPI

Article

Lyndon Factorization Algorithms for Small Alphabets and Run-Length Encoded Strings [†]

Sukhpal Singh Ghuman [1], Emanuele Giaquinta [2] and Jorma Tarhio [3,*]

[1] Faculty of Applied Science & Technology, Sheridan College, 7899 McLaughlin Road, Brampton, ON L6Y 5H9, Canada; sukhpal.ghuman@sheridancollege.ca
[2] F-Secure Corporation, P.O.B. 24, FI-00181 Helsinki, Finland; emanuele.giaquinta@f-secure.com
[3] Department of Computer Science, Aalto University, P.O.B. 15400, FI-00076 Aalto, Finland
[*] Correspondence: jorma.tarhio@aalto.fi

[†] This paper is an extended version of our paper: Ghuman, S.S.; Giaquinta, E.; Tarhio, J. Alternative algorithms for Lyndon factorization. In Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, 1–3 September 2014; pp. 169–178.

Received: 24 May 2019; Accepted: 17 June 2019; Published: 21 June 2019

Abstract: We present two modifications of Duval's algorithm for computing the Lyndon factorization of a string. One of the algorithms has been designed for strings containing runs of the smallest character. It works best for small alphabets and it is able to skip a significant number of characters of the string. Moreover, it can be engineered to have linear time complexity in the worst case. When there is a run-length encoded string R of length ρ, the other algorithm computes the Lyndon factorization of R in $O(\rho)$ time and in constant space. It is shown by experimental results that the new variations are faster than Duval's original algorithm in many scenarios.

Keywords: Lyndon factorization; string algorithms; run-length encoding

1. Introduction

A string w is a rotation of another string w' if $w = uv$ and $w' = vu$, for some strings u and v. A string is a Lyndon word if it is lexicographically smaller than all its proper rotations. Chen, Fox and Lyndon [1] introduced the unique factorization of a string in Lyndon words such that the sequence of factors is nonincreasing according to the lexicographical order. The Lyndon factorization is a key structure in a method for sorting the suffixes of a text [2], which is applied in the construction of the Burrows-Wheeler transform and the suffix array, as well as in the bijective variant of the Burrows-Wheeler transform [3,4]. The Burrows-Wheeler transform is an invertible transformation of a string, based on sorting of its rotations, while the suffix array is a lexicographically sorted array of the suffixes of a string. They are the groundwork for many indexing and data compression methods.

Duval's algorithm [5] computes the Lyndon factorization in linear time and in constant space. Various other solutions for computing the Lyndon factorization have been proposed in the past. A parallel algorithm [6] was presented by Apostolico and Crochemore, while Roh et al. described an external memory algorithm [7]. Recently, I et al. and Furuya et al. introduced algorithms to compute the Lyndon factorization of a string given in the grammar-compressed form and in the LZ78 encoding [8,9].

In this paper, we present two new variations of Duval's algorithm. The paper is an extended version of the conference paper [10]. The first algorithm has been designed for strings containing runs of the smallest character. It works best for small alphabets like the DNA alphabet {a, c, g, t} and it is able to skip a significant portion of the string. The second variation works for strings compressed with run-length encoding. In run-length encoding, maximal sequences in which the same data value occurs in many consecutive data elements (called runs) are stored as a pair of a single data value and a count. When there is a run-length encoded string R of length ρ, our algorithm computes the Lyndon factorization of R in $O(\rho)$ time and in constant space. This variation is thus preferable to Duval's algorithm when the strings are stored or maintained with run-length encoding. In our experiments, the new algorithms are considerably faster than the original one in the case of small alphabets, for both real and simulated data.

The rest of the paper is organized as follows. Section 2 defines background concepts Section 3 presents Duval's algorithm, Sections 4 and 5 introduce our variations of Duval's algorithm, Section 6 shows the results of our practical experiments, and the discussion of Section 7 concludes the article.

2. Basic Definitions

Let Σ be a finite ordered alphabet of σ symbols and let Σ^* be the set of words (strings) over Σ ordered by lexicographic order. In this paper, we use the terms string, sequence, and word interchangeably. The empty word ε is a word of length 0. Let Σ^+ be equal to $\Sigma^* \setminus \{\varepsilon\}$. Given a word w, we denote with $|w|$ the length of w and with $w[i]$ the i-th symbol of w, for $0 \leq i < |w|$. The concatenation of two words u and v is denoted by uv. Given two words u and v, v is a substring of u if there are indices $0 \leq i, j < |u|$ such that $v = u[i]...u[j]$. If $i = 0$ ($j = |u| - 1$) then v is a prefix (suffix) of u. The substring $u[i]...u[j]$ of u is denoted by $u[i..j]$, and for $i > j$ $u[i..j] = \varepsilon$. We denote by u^k the concatenation of k u's, for $u \in \Sigma^+$ and $k \geq 1$. The longest border of a word w, denoted with $\beta(w)$, is the longest proper prefix of w which is also a suffix of w. Let $lcp(w, w')$ denote the length of the longest common prefix of words w and w'. We write $w < w'$ if either $lcp(w, w') = |w| < |w'|$, i.e., if w is a proper prefix of w', or if $w[lcp(w, w')] < w'[lcp(w, w')]$. For any $0 \leq i < |w|$, $\text{ROT}(w, i) = w[i..|w| - 1]w[0..i - 1]$ is a rotation of w. A Lyndon word is a word w such that $w < \text{ROT}(w, i)$, for $1 \leq i < |w|$. Given a Lyndon word w, the following properties hold:

1. $|\beta(w)| = 0$;
2. either $|w| = 1$ or $w[0] < w[|w| - 1]$.

Both properties imply that no word a^k, for $a \in \Sigma$, $k \geq 2$, is a Lyndon word. The following result is due to Chen, Fox and Lyndon [11]:

Theorem 1. *Any word w admits a unique factorization $CFL(w) = w_1, w_2, \ldots, w_m$, such that w_i is a Lyndon word, for $1 \leq i \leq m$, and $w_1 \geq w_2 \geq \ldots \geq w_m$.*

The interval of positions in w of the factor w_i in $CFL(w) = w_1, w_2, \ldots, w_m$ is $[a_i, b_i]$, where $a_i = \sum_{j=1}^{i-1} |w_j|$, $b_i = \sum_{j=1}^{i} |w_j| - 1$, for $i = 1, \ldots, m$. We assume the following property:

Property 1. *The output of an algorithm that, given a word w, computes the factorization $CFL(w)$ is the sequence of intervals of positions of the factors in $CFL(w)$.*

The run-length encoding (RLE) of a word w, denoted by $\text{RLE}(w)$, is a sequence of pairs (runs) $\langle (c_1, l_1), (c_2, l_2,), \ldots, (c_\rho, l_\rho) \rangle$ such that $c_i \in \Sigma$, $l_i \geq 1$, $c_i \neq c_{i+1}$ for $1 \leq i < \rho$, and $w = c_1^{l_1} c_2^{l_2} \ldots c_\rho^{l_\rho}$. The interval of positions in w of the run (c_i, l_i) is $[a_i^{rle}, b_i^{rle}]$ where $a_i^{rle} = \sum_{j=1}^{i-1} l_j$, $b_i^{rle} = \sum_{j=1}^{i} l_j - 1$.

3. Duval's Algorithm

In this section we briefly describe Duval's algorithm for the computation of the Lyndon factorization of a word. Let L be the set of Lyndon words and let

$$P = \{w \mid w \in \Sigma^+ \text{ and } w\Sigma^* \cap L \neq \emptyset\},$$

be the set of nonempty prefixes of Lyndon words. Let also $P' = P \cup \{c^k \mid k \geq 2\}$, where c is the maximum symbol in Σ. Duval's algorithm is based on the following Lemmas, proved in [5]:

Lemma 1. *Let $w \in \Sigma^+$ and w_1 be the longest prefix of $w = w_1w'$ which is in L. We have $CFL(w) = w_1CFL(w')$.*

Lemma 2. $P' = \{(uv)^k u \mid u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ *and* $uv \in L\}$.

Lemma 3. *Let $w = (uav')^k u$, with $u, v' \in \Sigma^*$, $a \in \Sigma$, $k \geq 1$ and $uav' \in L$. The following propositions hold:*

1. *For $a' \in \Sigma$ and $a > a'$, $wa' \notin P'$;*
2. *For $a' \in \Sigma$ and $a < a'$, $wa' \in L$;*
3. *For $a' = a$, $wa' \in P' \setminus L$.*

Lemma 1 states that the computation of the Lyndon factorization of a word w can be carried out by computing the longest prefix w_1 of $w = w_1w'$ which is a Lyndon word and then recursively restarting the process from w'. Lemma 2 states that the nonempty prefixes of Lyndon words are all of the form $(uv)^k u$, where $u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ and $uv \in L$. By the first property of Lyndon words, the longest prefix of $(uv)^k u$ which is in L is uv. Hence, if we know that $w = (uv)^k uav'$, $(uv)^k u \in P'$ but $(uv)^k ua \notin P'$, then by Lemma 1 and by induction we have $CFL(w) = w_1w_2 \dots w_k\, CFL(uav')$, where $w_1 = w_2 = \dots = w_k = uv$. For example, if $w = abbabbaba$, we have $CFL(w) = abb\ abb\ CFL(aba)$, since $abbabbab \in P'$ while $abbabbaba \notin P'$.

Suppose that we have a procedure LF-NEXT(w,k) which computes, given a word w and an integer k, the pair (s,q) where s is the largest integer such that $w[k..k+s-1] \in L$ and q is the largest integer such that $w[k+is..k+(i+1)s-1] = w[k..k+s-1]$, for $i = 1, \dots, q-1$. The factorization of w can then be computed by iteratively calling LF-NEXT starting from position 0. When a given call to LF-NEXT returns, the factorization algorithm outputs the intervals $[k+is, k+(i+1)s-1]$, for $i = 0, \dots, q-1$, and restarts the factorization at position $k+qs$. Duval's algorithm implements LF-NEXT using Lemma 3, which explains how to compute, given a word $w \in P'$ and a symbol $a \in \Sigma$, whether $wa \in P'$, and thus makes it possible to compute the factorization using a left to right parsing. Note that, given a word $w \in P'$ with $|\beta(w)| = i$, we have $w[0..|w|-i-1] \in L$ and $w = (w[0..|w|-i-1])^q w[0..r-1]$ with $q = \lfloor \frac{|w|}{|w|-i} \rfloor$ and $r = |w| \mod (|w|-i)$. For example, if $w = abbabbab$, we have $|w| = 8$, $|\beta(w)| = 5$, $q = 2$, $r = 2$ and $w = (abb)^2 ab$. The code of Duval's algorithm is shown in Figure 1. The algorithm has $O(|w|)$-time and $O(1)$-space complexity.

```
LF-DUVAL($w$)
1.  $k \leftarrow 0$
2.  while $k < |w|$ do
3.      $(s,q) \leftarrow$ LF-NEXT$(w,k)$
4.      for $i \leftarrow 1$ to $q$ do
5.          output $(k, k+s-1)$
6.          $k \leftarrow k+s$
```

```
LF-NEXT($w,k$)
1.      $i \leftarrow k+1$
2.      $j \leftarrow k+2$
3.      while TRUE do
4.          if $j = |w|+1$ or $w[j-1] < w[i-1]$ then
5.              return$(j-i, \lfloor i/(j-i) \rfloor)$
6.          else
7.              if $w[j-1] > w[i-1]$ then
8.                  $i \leftarrow k+1$
9.              else
10.                 $i \leftarrow i+1$
11.             $j \leftarrow j+1$
```

Figure 1. Duval's algorithm to compute the Lyndon factorization of a string.

The following is an alternative formulation of Duval's algorithm by I et al. [8]:

Lemma 4. *Let $j > 0$ be any position of a string w such that $w < w[i..|w|-1]$ for any $0 < i \leq j$ and $lcp(w, w[j..|w|-1]) \geq 1$. Then, $w < w[k..|w|-1]$ also holds for any $j < k \leq j + lcp(w, w[j..|w|-1])$.*

Lemma 5. *Let w be a string with $CFL(w) = w_1, w_2, \dots, w_m$. It holds that $|w_1| = \min\{j \mid w[j..|w|-1] < w\}$ and $w_1 = w_2 = \dots = w_q = w[0..|w_1|-1]$, where $q = 1 + \lfloor lcp(w, w[|w_1|..|w|-1])/|w_1| \rfloor$.*

For example, if $w = abbabbaba$, we have $\min\{j \mid w[j..8] < w\} = 3$, $lcp(w, w[3..8]) = 5$, and $q = 2$. Based on these Lemmas, the procedure LF-NEXT can be implemented by initializing $j \leftarrow k+1$ and executing the following steps: (1) compute $h \leftarrow lcp(w[k..|w|-1], w[j..|w|-1])$. (2) if $j+h < |w|$ and $w[k+h] < w[j+h]$ set $j \leftarrow j+h+1$ and repeat step 1; otherwise return the pair $(j, 1+\lfloor h/j \rfloor)$. It is not hard to verify that, if the lcp values are computed using symbol comparisons, then this procedure corresponds to the one used by Duval's original algorithm.

4. Improved Algorithm for Small Alphabets

Let w be a word over an alphabet Σ with $CFL(w) = w_1, w_2, \ldots, w_m$ and let $\bar{c}$ be the smallest symbol in Σ. Suppose that there exists $k \geq 2, i \geq 1$ such that $\bar{c}^k$ is a prefix of w_i. If the last symbol of w is not $\bar{c}$, then by Theorem 1 and by the properties of Lyndon words, $\bar{c}^k$ is a prefix of each of $w_{i+1}, w_{i+2}, \ldots, w_m$. This property can be exploited to devise an algorithm for Lyndon factorization that can potentially skip symbols. Note that we assume Property 1, i.e., the output of the algorithm is the sequence of *intervals* of the factors in $CFL(w)$, as otherwise we have to read all the symbols of w to output $CFL(w)$. Our algorithm is based on the alternative formulation of Duval's algorithm by I et al. [8]. Given a set of strings $\mathcal{P}$, let $Occ_{\mathcal{P}}(w)$ be the set of all (starting) positions in w corresponding to occurrences of the strings in $\mathcal{P}$. We start with the following Lemmas:

Lemma 6. *Let w be a word and let $s = \max\{i \mid w[i] > \bar{c}\} \cup \{-1\}$. Then, we have $CFL(w) = CFL(w[0..s])CFL(\bar{c}^{(|w|-1-s)})$.*

Proof. If $s = -1$ or $s = |w|-1$ the Lemma plainly holds. Otherwise, Let w_i be the factor in $CFL(w)$ such that s belongs to $[a_i, b_i]$, the interval of w_i. To prove the claim we have to show that $b_i = s$. Suppose by contradiction that $s < b_i$, which implies $|w_i| \geq 2$. Then, $w_i[|w_i|-1] = \bar{c}$, which contradicts the second property of Lyndon words. □

For example, if $w = abaabaabbaabaa$, we have $CFL(w) = CFL(abaabaabbaab)\ CFL(aa)$.

Lemma 7. *Let w be a word such that $\bar{c}\bar{c}$ occurs in it and let $s = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. Then, we have $CFL(w) = CFL(w[0..s-1])CFL(w[s..|w|-1])$.*

Proof. Let w_i be the factor in $CFL(w)$ such that s belongs to $[a_i, b_i]$, the interval of w_i. To prove the claim we have to show that $a_i = s$. Suppose by contradiction that $s > a_i$, which implies $|w_i| \geq 2$. If $s = b_i$ then $w_i[|w_i|-1] = \bar{c}$, which contradicts the second property of Lyndon words. Otherwise, since w_i is a Lyndon word it must hold that $w_i < \text{ROT}(w_i, s-a_i)$. This implies at least that $w_i[0] = w_i[1] = \bar{c}$, which contradicts the hypothesis that s is the smallest element in $Occ_{\{\bar{c}\bar{c}\}}(w)$. □

For example, if $w = abaabaabbaab$, we have $CFL(w) = CFL(ab)\ CFL(aabaabbaab)$.

Lemma 8. *Let w be a word such that $w[0] = w[1] = \bar{c}$ and $w[|w|-1] \neq \bar{c}$. Let r be the smallest position in w such that $w[r] \neq \bar{c}$. Let also $\mathcal{P} = \{\bar{c}^r c \mid c \leq w[r]\}$. Then we have*

$$b_1 = \min\{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w|-1] < w\} \cup \{|w|\} - 1\,,$$

where b_1 is the ending position of factor w_1.

Proof. By Lemma 5 we have that $b_1 = \min\{s \mid w[s..|w|-1] < w\} - 1$. Since $w[0..r-1] = \bar{c}^r$ and $|w| \geq r+1$, for any string v such that $v < w$ we must have that either $v[0..r] \in \mathcal{P}$, if $|v| \geq r+1$, or $v = \bar{c}^{|v|}$ otherwise. Since $w[|w|-1] \neq \bar{c}$, the only position s that satisfies $w[s..|w|-1] = \bar{c}^{|w|-s}$ is $|w|$, corresponding to the empty word. Hence,

$$\{s \mid w[s..|w|-1] < w\} = \{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w|-1] < w\} \cup \{|w|\}.$$

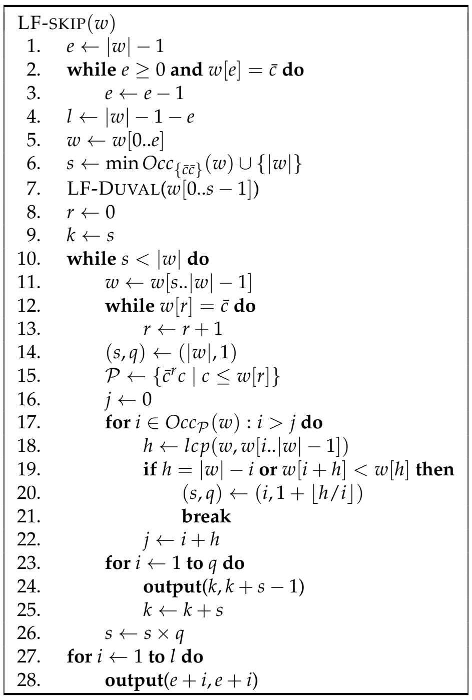
□

For example, if $w = aabaabbaab$, we have $\mathcal{P} = \{aaa, aab\}$, $Occ_{\mathcal{P}}(w) = \{0, 3, 7\}$ and $b_1 = 6$. Based on these Lemmas, we can devise a faster factorization algorithm for words containing runs of $\bar{c}$. The key idea is that, using Lemma 8, it is possible to skip symbols in the computation of b_1, if a suitable string matching algorithm is used to compute $Occ_{\mathcal{P}}(w)$. W.l.o.g. we assume that the last symbol of w is different from $\bar{c}$. In the general case, by Lemma 6, we can reduce the factorization of w to the one of its longest prefix with last symbol different from $\bar{c}$, as the remaining suffix is a concatenation of $\bar{c}$ symbols, whose factorization is a sequence of factors equal to $\bar{c}$. Suppose that $\bar{c}\bar{c}$ occurs in w. By Lemma 7 we can split the factorization of w in $CFL(u)$ and $CFL(v)$ where $uv = w$ and $|u| = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. The factorization of $CFL(u)$ can be computed using Duval's original algorithm.

Concerning v, let $r = \min\{i \mid v[i] \neq \bar{c}\}$. By definition $v[0] = v[1] = \bar{c}$ and $v[|v| - 1] \neq \bar{c}$, and we can apply Lemma 8 on v to find the ending position s of the first factor in $CFL(v)$, i.e., $\min\{i \in Occ_{\mathcal{P}}(v) \mid v[i..|v| - 1] < v\}$, where $\mathcal{P} = \{\bar{c}^r c \mid c \leq v[r]\}$. To this end, we iteratively compute $Occ_{\mathcal{P}}(v)$ until either a position i is found that satisfies $v[i..|v| - 1] < v$ or we reach the end of the string. Let $h = lcp(v, v[i..|v| - 1])$, for a given $i \in Occ_{\mathcal{P}}(v)$. Observe that $h \geq r$ and, if $v < v[i..|v| - 1]$, then, by Lemma 4, we do not need to verify the positions $i' \in Occ_{\mathcal{P}}(v)$ such that $i' \leq i + h$. The computation of $Occ_{\mathcal{P}}(v)$ can be performed by using either an algorithm for multiple string matching for the set of patterns $\mathcal{P}$ or an algorithm for single string matching for the pattern $\bar{c}^r$, since $Occ_{\mathcal{P}}(v) \subseteq Occ_{\bar{c}^r}(v)$. Note that the same algorithm can also be used to compute $\min Occ_{\bar{c}\bar{c}}(w)$ in the first phase.

Given that all the patterns in $\mathcal{P}$ differ in the last symbol only, we can express $\mathcal{P}$ more succinctly using a character class for the last symbol and match this pattern using a string matching algorithm that supports character classes, such as the algorithms based on bit-parallelism. In this respect, SBNDM2 [12], a variation of the BNDM algorithm [13] is an ideal choice, as it is sublinear on average. However, this method is preferable only if $r + 1$ is less than or equal to the machine word size in bits.

Let $h = lcp(v, v[s..|v| - 1])$ and $q = 1 + \lfloor h/s \rfloor$. Based on Lemma 5, the algorithm then outputs the intervals of the factors $v[(i - 1)s..is - 1]$, for $i = 1, \ldots q$, and iteratively applies the above method on $v' = v[sq..|v| - 1]$. It is not hard to verify that, if $v' \neq \varepsilon$, then $|v'| \geq r + 1$, $v'[0..r - 1] = \bar{c}$ and $v'[|v'| - 1] \neq \bar{c}$, and so Lemma 8 can be used on v'. The code of the algorithm, named LF-SKIP, is shown in Figure 2. The computation of the value $r' = \min\{i \mid v'[i] \neq \bar{c}\}$ for v' takes advantage of the fact that $v'[0..r - 1] = \bar{c}$, so as to avoid useless comparisons.



```
LF-SKIP($w$)
1.  $e \leftarrow |w| - 1$
2.  while $e \geq 0$ and $w[e] = \bar{c}$ do
3.      $e \leftarrow e - 1$
4.  $l \leftarrow |w| - 1 - e$
5.  $w \leftarrow w[0..e]$
6.  $s \leftarrow \min Occ_{\{\bar{c}\bar{c}\}}(w) \cup \{|w|\}$
7.  LF-DUVAL($w[0..s - 1]$)
8.  $r \leftarrow 0$
9.  $k \leftarrow s$
10. while $s < |w|$ do
11.     $w \leftarrow w[s..|w| - 1]$
12.     while $w[r] = \bar{c}$ do
13.         $r \leftarrow r + 1$
14.     $(s, q) \leftarrow (|w|, 1)$
15.     $\mathcal{P} \leftarrow \{\bar{c}^r c \mid c \leq w[r]\}$
16.     $j \leftarrow 0$
17.     for $i \in Occ_{\mathcal{P}}(w) : i > j$ do
18.         $h \leftarrow lcp(w, w[i..|w| - 1])$
19.         if $h = |w| - i$ or $w[i + h] < w[h]$ then
20.             $(s, q) \leftarrow (i, 1 + \lfloor h/i \rfloor)$
21.             break
22.         $j \leftarrow i + h$
23.     for $i \leftarrow 1$ to $q$ do
24.         output($k, k + s - 1$)
25.         $k \leftarrow k + s$
26.     $s \leftarrow s \times q$
27. for $i \leftarrow 1$ to $l$ do
28.     output($e + i, e + i$)
```

Figure 2. The algorithm to compute the Lyndon factorization that can potentially skip symbols.

If the total time spent for the iteration over the sets $Occ_{\mathcal{P}}(v)$ is $O(|w|)$, the worst case time complexity of LF-SKIP is linear. To see why, it is enough to observe that the positions i for which LF-SKIP verifies if $v[i..|v|-1] < v$ are a subset of the positions verified by the original algorithm. Indeed, given a string w satisfying the conditions of Lemma 8, for any position $i \in Occ_{\mathcal{P}}$ there is no $i' \in \{0, 1, \dots, |w|-1\} \setminus Occ_{\mathcal{P}}$ such that $i'+1 \le i \le i' + lcp(w, w[i'..|w|-1])$. Hence, the only way Duval's algorithm can skip a position $i \in Occ_{\mathcal{P}}$ using Lemma 4 is by means of a smaller position i' belonging to $Occ_{\mathcal{P}}$, which implies that the algorithms skip or verify the same positions in $Occ_{\mathcal{P}}$.

5. Computing the Lyndon Factorization of a Run-Length Encoded String

In this section we present an algorithm to compute the Lyndon factorization of a string given in RLE form. The algorithm is based on Duval's original algorithm and on a combinatorial property between the Lyndon factorization of a string and its RLE, and has $O(\rho)$-time and $O(1)$-space complexity, where ρ is the length of the RLE. We start with the following Lemma:

Lemma 9. *Let w be a word over Σ and let $w_1, w_2, \dots, w_m$ be its Lyndon factorization. For any $1 \le i \le |\text{RLE}(w)|$, let $1 \le j, k \le m$, $j \le k$, such that $a_i^{rle} \in [a_j, b_j]$ and $b_i^{rle} \in [a_k, b_k]$. Then, either $j = k$ or $|w_j| = |w_k| = 1$.*

Proof. Suppose by contradiction that $j < k$ and either $|w_j| > 1$ or $|w_k| > 1$. By definition of j, k, we have $w_j \ge w_k$. Moreover, since both $[a_j, b_j]$ and $[a_k, b_k]$ overlap with $[a_i^{rle}, b_i^{rle}]$, we also have $w_j[|w_j|-1] = w_k[0]$. If $|w_j| > 1$, then, by definition of w_j, we have $w_j[0] < w_j[|w_j|-1] = w_k[0]$. Instead, if $|w_k| > 1$ and $|w_j| = 1$, we have that w_j is a prefix of w_k. Hence, in both cases we obtain $w_j < w_k$, which is a contradiction. □

The consequence of this Lemma is that a run of length l in the RLE is either contained in *one* factor of the Lyndon factorization, or it corresponds to l unit-length factors. Formally:

Corollary 1. *Let w be a word over Σ and let $w_1, w_2, \dots, w_m$ be its Lyndon factorization. Then, for any $1 \le i \le |\text{RLE}(w)|$, either there exists w_j such that $[a_i^{rle}, b_i^{rle}]$ is contained in $[a_j, b_j]$ or there exist l_i factors $w_j, w_{j+1}, \dots, w_{j+l_i-1}$ such that $|w_{j+k}| = 1$ and $a_{j+k} \in [a_i^{rle}, b_i^{rle}]$, for $0 \le k < l_i$.*

This property can be exploited to obtain an algorithm for the Lyndon factorization that runs in $O(\rho)$ time. First, we introduce the following definition:

Definition 1. *A word w is a LR word if it is either a Lyndon word or it is equal to a^k, for some $a \in \Sigma$, $k \ge 2$. The LR factorization of a word w is the factorization in LR words obtained from the Lyndon factorization of w by merging in a single factor the maximal sequences of unit-length factors with the same symbol.*

For example, the LR factorization of *cctgccaa* is $\langle cctg, cc, aa \rangle$. Observe that this factorization is a (reversible) encoding of the Lyndon factorization. Moreover, in this encoding it holds that each run in the RLE is contained in one factor and thus the size of the LR factorization is $O(\rho)$. Let L' be the set of LR words. Suppose that we have a procedure LF-RLE-NEXT(R, k) which computes, given an RLE sequence R and an integer k, the pair (s, q) where s is the largest integer such that $c_k^{l_k} \dots c_{k+s-1}^{l_{k+s-1}} \in L'$ and q is the largest integer such that $c_{k+is}^{l_{k+is}} \dots c_{k+(i+1)s-1}^{l_{k+(i+1)s-1}} = c_k^{l_k} \dots c_{k+s-1}^{l_{k+s-1}}$, for $i = 1, \dots, q-1$. Observe that, by Lemma 9, $c_k^{l_k} \dots c_{k+s-1}^{l_{k+s-1}}$ is the longest prefix of $c_k^{l_k} \dots c_\rho^{l_\rho}$ which is in L', since otherwise the run (c_{k+s}, l_{k+s}) would span two factors in the LR factorization of $c_k^{l_k} \dots c_\rho^{l_\rho}$. This implies that the pair (s, q) returned by LF-RLE-NEXT(R, k) satisfies

$$\text{LF-NEXT}(c_k^{l_k} \dots c_\rho^{l_\rho}, 0) = \begin{cases} (\sum_{i=0}^{s-1} l_{k+i}, q) & \text{if } s > 1, \\ (1, l_k) & \text{otherwise.} \end{cases}$$

Based on Lemma 1, the factorization of R can then be computed by iteratively calling LF-RLE-NEXT starting from position 0. When a given call to LF-RLE-NEXT returns, the factorization algorithm outputs the intervals $[k + is, k + (i + 1)s - 1]$ in R, for $i = 0, \dots, q - 1$, and restarts the factorization at position $k + qs$.

We now present the LF-RLE-NEXT algorithm. Analogously to Duval's algorithm, it reads the RLE sequence from left to right maintaining two integers, j and ℓ, which satisfy the following invariant:

$$\begin{aligned} & c_k^{l_k} \dots c_{j-1}^{l_{j-1}} \in P'; \\ & \ell = \begin{cases} |\mathrm{RLE}(\beta(c_k^{l_k} \dots c_{j-1}^{l_{j-1}}))| & \text{if } j - k > 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

The integer j, initialized to $k + 1$, is the index of the next run to read and is incremented at each iteration until either $j = |R|$ or $c_k^{l_k} \dots c_{j-1}^{l_{j-1}} \notin P'$. The integer ℓ, initialized to 0, is the length in runs of the longest border of $c_k^{l_k} \dots c_{j-1}^{l_{j-1}}$, if $c_k^{l_k} \dots c_{j-1}^{l_{j-1}}$ spans at least two runs, and equal to 0 otherwise. For example, in the case of the word ab^2ab^2ab we have $\beta(ab^2ab^2ab) = ab^2ab$ and $\ell = 4$. Let $i = k + \ell$. In general, if $\ell > 0$, we have

$$\begin{aligned} & l_{j-1} \le l_{i-1}, l_k \le l_{j-\ell}, \\ & \beta(c_k^{l_k} \dots c_{j-1}^{l_{j-1}}) = c_k^{l_k} c_{k+1}^{l_{k+1}} \dots c_{i-2}^{l_{i-2}} c_{i-1}^{l_{j-1}} = c_{j-\ell}^{l_k} c_{j-\ell+1}^{l_{j-\ell+1}} \dots c_{j-2}^{l_{j-2}} c_{j-1}^{l_{j-1}}. \end{aligned}$$

Note that the longest border may not fully cover the last (first) run of the corresponding prefix (suffix). Such the case is for example for the word ab^2a^2b. However, since $c_k^{l_k} \dots c_{j-1}^{l_{j-1}} \in P'$ it must hold that $l_{j-\ell} = l_k$, i.e., the first run of the suffix is fully covered. Let

$$z = \begin{cases} 1 & \text{if } \ell > 0 \land l_{j-1} < l_{i-1}, \\ 0 & \text{otherwise.} \end{cases}$$

Informally, the integer z is equal to 1 if the longest border of $c_k^{l_k} \dots c_{j-1}^{l_{j-1}}$ does not fully cover the run (c_{i-1}, l_{i-1}). By 1 we have that $c_k^{l_k} \dots c_{j-1}^{l_{j-1}}$ can be written as $(uv)^q u$, where

$$\begin{aligned} & q = \lfloor \tfrac{j-k-z}{j-i} \rfloor, r = z + (j - k - z) \mod (j - i), \\ & u = c_{j-r}^{l_{j-r}} \dots c_{j-1}^{l_{j-1}}, uv = c_k^{l_k} \dots c_{j-\ell-1}^{l_{j-\ell-1}} = c_{i-r}^{l_{i-r}} \dots c_{j-r-1}^{l_{j-r-1}}, \\ & uv \in L' \end{aligned}$$

For example, in the case of the word ab^2ab^2ab, for $k = 0$, we have $j = 6, i = 4, q = 2, r = 2$. The algorithm is based on the following Lemma:

Lemma 10. *Let j, ℓ be such that invariant 1 holds and let $s = i - z$. Then, we have the following cases:*

1. *If $c_j < c_s$ then $c_k^{l_k} \dots c_j^{l_j} \notin P'$;*
2. *If $c_j > c_s$ then $c_k^{l_k} \dots c_j^{l_j} \in L'$ and 1 holds for $j + 1$, $\ell' = 0$;*

Moreover, if $z = 0$, we also have:

3. *If $c_j = c_i$ and $l_j \le l_i$, then $c_k^{l_k} \dots c_j^{l_j} \in P'$ and 1 holds for $j + 1$, $\ell' = \ell + 1$;*
4. *If $c_j = c_i$ and $l_j > l_i$, either $c_j < c_{i+1}$ and $c_k^{l_k} \dots c_j^{l_j} \notin P'$ or $c_j > c_{i+1}$, $c_k^{l_k} \dots c_j^{l_j} \in L'$ and 1 holds for $j + 1$, $\ell' = 0$.*

Proof. The idea is the following: we apply Lemma 3 with the word $(uv)^q u$ as defined above and symbol c_j. Observe that c_j is compared with symbol $v[0]$, which is equal to $c_{k+r-1} = c_{i-1}$ if $z = 1$ and to $c_{k+r} = c_i$ otherwise.

First note that, if $z = 1$, $c_j \neq c_{i-1}$, since otherwise we would have $c_{j-1} = c_{i-1} = c_j$. In the first three cases, we obtain the first, second and third proposition of Lemma 3, respectively, for the word $c_k^{l_k} \dots c_{j-1}^{l_{j-1}} c_j$. Independently of the derived proposition, it is easy to verify that the same proposition also holds for $c_k^{l_k} \dots c_{j-1}^{l_{j-1}} c_j^m$, $m \leq l_j$. Consider now the fourth case. By a similar reasoning, we have that the third proposition of Lemma 3 holds for $c_k^{l_k} \dots c_j^{l_i}$. If we then apply Lemma 3 to $c_k^{l_k} \dots c_j^{l_i}$ and c_j, c_j is compared to c_{i+1} and we must have $c_j \neq c_{i+1}$ as otherwise $c_i = c_j = c_{i+1}$. Hence, either the first (if $c_j < c_{i+1}$) or the second (if $c_j > c_{i+1}$) proposition of Lemma 3 must hold for the word $c_k^{l_k} \dots c_j^{l_i+1}$. □

We prove by induction that invariant 1 is maintained. At the beginning, the variables j and ℓ are initialized to $k+1$ and 0, respectively, so the base case trivially holds. Suppose that the invariant holds for j, ℓ. Then, by Lemma 10, either $c_k^{l_k} \dots c_j^{l_j} \notin P'$ or it follows that the invariant also holds for $j+1, \ell'$, where ℓ' is equal to $\ell + 1$, if $z = 0$, $c_j = c_i$ and $l_j \leq l_i$, and to 0 otherwise. When $c_k^{l_k} \dots c_j^{l_j} \notin P'$ the algorithm returns the pair $(j-i, q)$, i.e., the length of uv and the corresponding exponent.

The code of the algorithm is shown in Figure 3. We now prove that the algorithm runs in $O(\rho)$ time. First, observe that, by definition of LR factorization, the for loop at line 4 is executed $O(\rho)$ times. Suppose that the number of iterations of the while loop at line 2 is n and let $k_1, k_2, \dots, k_{n+1}$ be the corresponding values of k, with $k_1 = 0$ and $k_{n+1} = |R|$. We now show that the s-th call to LF-RLE-NEXT performs less than $2(k_{s+1} - k_s)$ iterations, which will yield $O(\rho)$ number of iterations in total. This analysis is analogous to the one used by Duval. Suppose that i', j' and z' are the values of i, j and z at the end of the s-th call to LF-RLE-NEXT. The number of iterations performed during this call is equal to $j' - k_s$. We have $k_{s+1} = k_s + q(j' - i')$, where $q = \lfloor \frac{j'-k_s-z}{j-i'} \rfloor$, which implies $j' - k_s < 2(k_{s+1} - k_s) + 1$, since, for any positive integers x, y, $x < 2\lfloor x/y \rfloor y$ holds.

```
LF-RLE(R)
1.  k ← 0
2.  while k < |R| do
3.      (s, q) ← LF-RLE-NEXT(R, k)
4.      for i ← 1 to q do
5.          output (k, k + s − 1)
6.          k ← k + s
```

```
LF-RLE-NEXT(R = ⟨(c_1, l_1), ..., (c_ρ, l_ρ)⟩, k)
1.  i ← k
2.  j ← k + 1
3.  while TRUE do
4.      if i > k and l_{j−1} < l_{i−1} then
5.          z ← 1
6.      else z ← 0
7.      s ← i − z
8.      if j = |R| or c_j < c_s or
9.          (c_j = c_s and l_j > l_s and c_j < c_{s+1}) then
10.     return (j − i, ⌊(j − k − z)/(j − i)⌋)
11.     else
12.         if c_j > c_s or l_j > l_s then
13.             i ← k
14.         else
15.             i ← i + 1
16.         j ← j + 1
```

Figure 3. The algorithm to compute the Lyndon factorization of a run-length encoded string.

6. Experimental Results

We tested extensively the algorithms LF-DUVAL, LF-SKIP, and LF-RLE. In addition, we also tested variations of LF-DUVAL and LF-SKIP, denoted as LF-DUVAL2 and LF-SKIP2. LF-DUVAL2 performs an if-test

$$\textbf{if } w[j-1] = w[i-1] \textbf{ then}$$

which is always true in line 9 of LF-NEXT. This form, which is advantageous for compiler optimization, can be justified by the formulation of the original algorithm [5] where there is a three branch test of $w[j-1]$ and $w[i-1]$. LF-SKIP2, after finding the first $\bar{c}^r$, searches for $\bar{c}^r$ until $\bar{c}^{r+1}$ is found, whereas LF-SKIP searches for $\bar{c}^r x$ where x is a character class.

The experiments were run on Intel Core i7-4578U with 3 GHz clock speed and 16 GB RAM. The algorithms were written in the C programming language and compiled with gcc 5.4.0 using the O3 optimization level.

Testing LF-SKIP. At first we tested the variations of LF-SKIP against the variations of LF-DUVAL. The texts were random sequences of 5 MB symbols. For each alphabet size $\sigma = 2, 4, \ldots, 256$ we generated 100 sequences with a uniform distribution, and each run with each sequence was repeated 500 times. The average run times are given in Table 1 which is shown in a graphical form in Figure 4.

Table 1. Run times in milliseconds on random sequences (5 MB) with a uniform distribution of a varying alphabet size.

σ	LF-DUVAL	LF-DUVAL2	LF-SKIP	LF-SKIP2
2	14.6	21.9	2.5	1.5
4	14.6	14.9	1.6	1.1
8	14.7	9.1	1.3	1.1
16	14.7	6.4	1.3	1.2
32	14.7	5.0	1.4	1.6
64	14.7	4.3	1.7	2.3
128	14.7	4.0	2.0	3.2
192	14.6	3.8	1.7	3.7
256	14.6	3.8	2.6	4.1

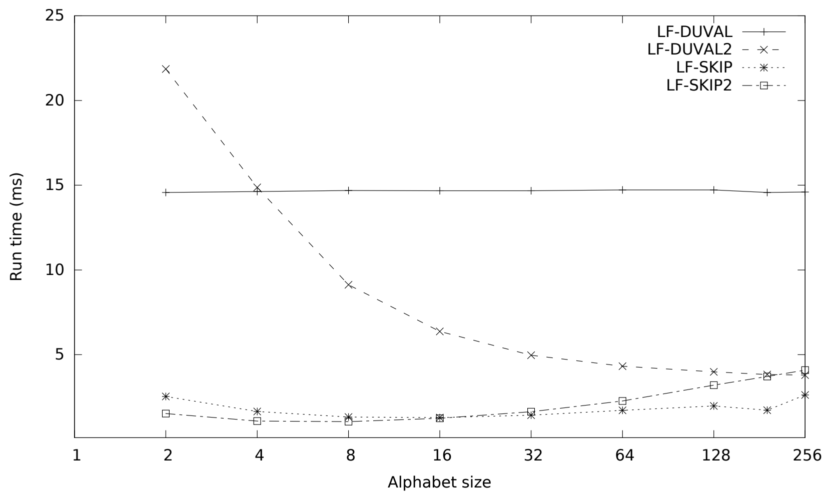


Figure 4. Comparison of the algorithms on random sequences (5 MB) with a uniform distribution of a varying alphabet size.

LF-SKIP was faster than the best variation of LF-DUVAL for all tested values of σ. The speed-up was significant for small alphabets. LF-SKIP2 was faster than LF-SKIP for $\sigma \leq 16$ and slower for $\sigma > 16$.

The speed of LF-DUVAL did not depend on σ. LF-DUVAL2 became faster when the size of the alphabet grew. For large alphabets LF-DUVAL2 was faster than LF-DUVAL and for small alphabets the other way round. In additional refined experiments, $\sigma = 5$ was the threshold value. When we compiled LF-DUVAL and LF-DUVAL2 without optimization, both of the variations behaved in a similar way. So the better performance of LF-DUVAL2 for large alphabets is due to compiler optimization, possibly by cause of branch prediction.

We tested the variations of LF-SKIP also with longer random sequences of four characters up to 500 MB. The average speed did not essentially change when the sequence became longer.

In addition, we tested LF-SKIP and LF-SKIP2 with real texts. At first we did experiments with texts of natural language. Because runs are very short in a natural language and newline or some other control character is the smallest character, the benefit of LF-SKIP or LF-SKIP2 was marginal. If it were acceptable to relax the lexicographic order of the characters, some gain could be obtained. For example, LF-SKIP achieved the speed-up of 2 over LF-DUVAL2 in the case of the KJV Bible when '*l*' is the smallest character.

For the DNA sequence of fruitfly (15 MB), LF-SKIP2 was 20.3 times faster than LF-DUVAL. For the protein sequence of the saccharomyces cerevisiae (2.9 MB), LF-SKIP2 was 8.7 times faster than LF-DUVAL2. The run times on these biological sequences are shown in Table 2.

Table 2. Run times in milliseconds on two biological sequences.

	LF-DUVAL	LF-DUVAL2	LF-SKIP	LF-SKIP2
DNA (15 MB)	44.7	52.2	3.0	2.2
Protein (2.9 MB)	8.5	3.4	0.50	0.39

Testing LF-RLE. To assess the performance of the LF-RLE algorithm, we tested it together with LF-DUVAL, LF-DUVAL2 and LF-SKIP2 for random binary sequences of 5 MB with different probability distributions, so as to vary the number of runs in the sequence. The running time of LF-RLE does not include the time needed to compute the RLE of the sequence, i.e., we assumed that the sequence is given in the RLE form, since otherwise other algorithms are preferable. For each test we generated 100 sequences, and each run with each sequence was repeated 500 times. The average run times are given in Table 3 which is shown in a graphical form in Figure 5.

Table 3. Run times in milliseconds on random binary sequences (5 MB) with a skew distribution.

P(zero)	LF-DUVAL	LF-DUVAL2	LF-SKIP2	LF-RLE
0.05	14.6	5.7	1.4	0.70
0.10	14.7	7.8	1.1	1.3
0.20	14.7	12.4	1.0	2.4
0.30	14.8	17.4	1.2	3.2
...				
0.70	14.7	16.9	1.7	3.2
0.80	14.6	12.7	2.0	2.4
0.90	14.6	8.4	2.8	1.3
0.95	14.7	6.3	4.7	0.70

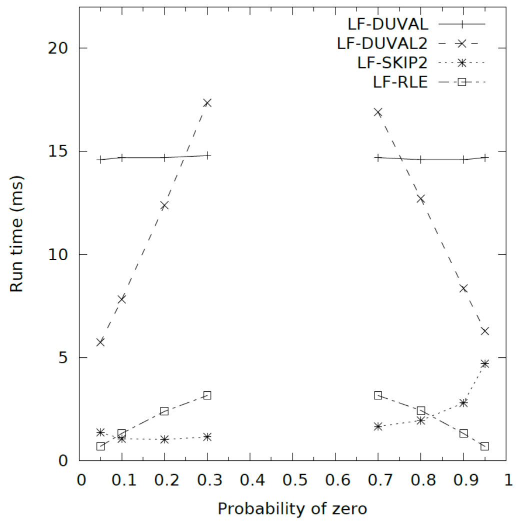


Figure 5. Comparison of the algorithms on random binary sequences (5 MB) with a skew distribution.

Table 3 shows that LF-RLE was the fastest for distributions $P(0) = 0.05$, 0.9, and 0.95. Table 3 also reveals that LF-RLE and LF-DUVAL2 worked symmetrically for distributions of zero and one, but LF-SKIP2 worked unsymmetrically which is due to the fact that LF-SKIP2 searches for the runs of the smallest character which was zero in this case.

In our tests the run time of LF-DUVAL was about 14.7 ms for all sequences of 5 MB. Thus LF-DUVAL is a better choice than LF-DUVAL2 for cases $P(0) = 0.3$ and 0.7.

7. Conclusions

We presented new variations of Duval's algorithm for computing the Lyndon factorization of a string. The first algorithm LF-SKIP was designed for strings containing runs of the smallest character in the alphabet and it is able to skip a significant portion of the characters of the string. The second algorithm LF-RLE is for strings compressed with run-length encoding and computes the Lyndon factorization of a run-length encoded string of length ρ in $O(\rho)$ time and constant space. Our experimental results show that these algorithms can offer a significant speed-up over Duval's original algorithm. Especially LF-SKIP is efficient in the case of biological sequences.

Author Contributions: Formal analysis, E.G.; Investigation, S.S.G.; Methodology, J.T.; Software, E.G. and J.T.; Supervision, J.T.; Writing—original draft, S.S.G. and E.G.; Writing—review & editing, J.T.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Chen, K.T.; Fox, R.H.; Lyndon, R.C. Free differential calculus. IV. The quotient groups of the lower central series. *Ann. Math.* **1958**, *68*, 81–95. [CrossRef]
2. Mantaci, S.; Restivo, A.; Rosone, G.; Sciortino, M. Sorting suffixes of a text via its Lyndon factorization. In Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, 2–4 September 2013; pp. 119–127.
3. Gil, J.Y.; Scott, D.A. A bijective string sorting transform. *arXiv* **2012**, arXiv:1201.3077.
4. Kufleitner, M. On bijective variants of the Burrows-Wheeler transform. In Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, 31 August–2 September 2009; pp. 65–79.
5. Duval, J.P. Factorizing words over an ordered alphabet. *J. Algorithms* **1983**, *4*, 363–381. [CrossRef]
6. Apostolico, A.; Crochemore, M. Fast parallel Lyndon factorization with applications. *Math. Syst. Theory* **1995**, *28*, 89–108. [CrossRef]
7. Roh, K.; Crochemore, M.; Iliopoulos, C.S.; Park, K. External memory algorithms for string problems. *Fundam. Inform.* **2008**, *84*, 17–32.
8. Tomohiro, I.; Nakashima, Y.; Inenaga, S.; Bannai, H.; Takeda, M. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.* **2016**, *656*, 215–224. [CrossRef]
9. Furuya, I.; Nakashima, Y.; Tomohiro, I.; Inenaga, S.; Bannai, H.; Takeda, M. Lyndon Factorization of Grammar Compressed Texts Revisited. In Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM 2018), Qingdao, China, 2–4 July 2018. [CrossRef]
10. Ghuman, S.S.; Giaquinta, E.; Tarhio, J. Alternative algorithms for Lyndon factorization. In Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, 1–3 September 2014; pp. 169–178.
11. Lothaire, M. *Combinatorics on Words*; Cambridge Mathematical Library, Cambridge University Press: Cambridge, UK, 1997.
12. Durian, B.; Holub, J.; Peltola, H.; Tarhio, J. Improving practical exact string matching. *Inf. Process. Lett.* **2010**, *110*, 148–152. [CrossRef]
13. Navarro, G.; Raffinot, M. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Exp. Algorithm* **2000**, *5*, 4. [CrossRef]