Lempel-Ziv Factorization in Linear-Time $O(1)$-Workspace for Constant Alphabets

Weijun LIU$^{[a]}$, Member

SUMMARY Computing the Lempel-Ziv Factorization (LZ77) of a string is one of the most important problems in computer science. Nowadays, it has been widely used in many applications such as data compression, text indexing and pattern discovery, and already become the heart of many file compressors like gzip and 7zip. In this paper, we show a linear time algorithm called Xone for computing the LZ77, which has the same space requirement with the previous best space requirement for linear time LZ77 factorization called BGone. Xone greatly improves the efficiency of BGone. Experiments show that the two versions of Xone: XoneT and XoneSA are about 27% and 31% faster than BGoneT and BGoneSA, respectively.

key words: Lempel-Ziv factorization, induced sorting, string, algorithm, linear time, data compression, suffix array

1. Introduction

Lempel-Ziv factorization (LZ77) [1] of a string has been widely used in string processing, genome sequence alignments [2], [3], text indexes [4], data compression and de-compression [5]–[8], etc. Especially, compression schemes based on LZ77 can be particularly good at processing the modern datasets with highly repetitive characteristics in recent years. To most of applications, computing LZ77 is a time and space bottleneck. Thus, the research on both time and space efficient LZ77 computation algorithms has become a hot pursuit [9]–[16].

There have been many algorithms for computing the LZ77 in linear time [9]–[11], [13]. All of them are based on the suffix array. For an input string of length $n$, these algorithms require at least $2n \log n$ bits of working space. And these algorithms mainly construct several auxiliary integer arrays of length $n$: the Inverse Suffix Array (ISA), the Longest Common Prefix (LCP) array [17] and the Longest Previous Factor (LPF) array [10]. Naturally, these earlier algorithms require at least $3n \log n$ bits of working space, i.e., three integer arrays of length $n$. The working space excludes the input string and the output factorization. Since the values in LCP and LPF are not required any more during computing the LZ77, some algorithms do not construct these arrays at the same time [12], [13], [18] and achieved better time and space complexity. To our knowledge, the current fastest linear time algorithms for computing LZ77 are KKP3, KKP2 and KKP1, which were proposed by Kärkkäinen et al. [12], and respectively use three, two and one auxiliary integer array. KKP3 is similar to $LZ_{BG}$ [13] and can be seen as a reorganization of it. The reorganization is effective because KKP3 is more smaller and runs faster. By taking advantage of the relationships between $NSV$, $PSV$ and $\Phi$ arrays, KKP2 reduces the total working space to two integer arrays. Although the working space is reduced, the speed is comparable to KKP3 for some input strings. KKP1 stores the suffix array on the disk and then reads it from the disk when computing NSV. Besides that, there is no significant difference between KKP1 and KKP2. So the total working space of KKP1 is still two integer arrays, one keeping in memory and the other keeping in disk.

KKP runs fast, however, time and space efficient algorithms for computing LZ77 are needed and very important [19]. Motivated by this, Goto and Bannai [18] proposed an algorithm called BGone which greatly improves the previous best space requirement for computing LZ77 in $O(n)$ time, and uses only $n \log n + O(\sigma \log n)$ bits of working space, i.e., one integer array of length $n$ plus $O(1)$ extra working space. BGone has two versions: BGoneT and BGoneSA. BGoneT computes LZ77 directly from the input string $T$. Unlike other LZ77 algorithms, it does not have to compute the suffix array in advance. BGoneSA computes LZ77 from the $S_A$ of $T$. Given $S_A$, BGoneSA is a better choice. For there are many suffix array construction algorithms in the industry, BGoneSA can freely choose the more suitable suffix array algorithms to compute the suffix array. The experimental results show that the speed of BGone is around two to three times slower than KKP3. The speed of BGone is a bit slow. Faster algorithms are very much required.

In recent years, computing LZ77 using small working space is a long-standing problem [20], and low-space algorithms is still a very important and hot research topic in many fields such as Massively Parallel Computation (MPC) [21], construction of AVL grammars [22] and suffix array for big genome data [23].

So, the motivation of this paper is to design more faster algorithms for computing LZ77. We present here an algorithm called Xone, which has two versions XoneT and XoneSA corresponding to BGoneT and BGoneSA respectively. Xone has the same time and space complexities with...
BGone, but runs much faster than BGone. In summary, the main contributions of this paper are as follows.

- We design an algorithm called XoneT to compute the LZ77 directly from the input string $T$ without constructing the suffix array. During computing $\Phi_{lms}$, we introduce the notion of L- and S-type positions of suffix array that can efficiently link all LMS-suffixes for induced sorting L-suffixes. Considering that in some cases the suffix array has been obtained, we provide another version of XoneT called XoneSA, which can compute the LZ77 from the suffix array. Both XoneT and XoneSA are space economical and run in linear time.
- We experimentally evaluate Xone on a variety of datasets against the others, such as KKP, BGone and BGtwo. Our results show that KKP3 is on average 1.64 faster than XoneT. However, the working space of Xone is only 1/3 that of KKP3. Xone has the same time and space complexities with BGone. But XoneT and XoneSA are more efficient and run around 27% and 31% faster than BGoneT and BGoneSA, respectively.

2. Notation and Basic Concepts

2.1 String

In this paper, we assume that $\Sigma$ is a constant alphabet of size $\sigma$. $T = T[1..n] = T[1]T[2]...T[n]$ is a string of length $n$, where $T[i]$ is the $i$-th character of $T$ and drawn from $\Sigma$. The length of a string $T$ is denoted by $|T|$ and $|T| = n$. For any $i$, $j (i < j) \in [1..n]$, we write $T[i..j]$ to denote the substring $T[i]T[i+1]...T[j]$ of length $j - i + 1$.

2.2 Suffix Array

For convenience, we assume that the last character of $T$ is the sentinel $\$, which is the unique lexicographically smallest character in $T$. For $i \in [1..n]$, we write $\text{suffix}(T, i)$ to denote the suffix of $T$, which is a substring that starts from $T[i]$ and ends at $T[n]$, i.e., $\text{suffix}(T, i) = T[i..n]$. Similarly, we write $\text{prefix}(T, i)$ to denote the prefix of $T$, which is a substring that starts from $T[1]$ and ends at $T[i]$, i.e., $\text{prefix}(T, i) = T[1..i]$. Let $\text{lcp}(i, j)$ be the length of the longest common prefix (LCP) of $\text{suffix}(T, i)$ and $\text{suffix}(T, j)$. For example, given $T = \$abb\$bbbbaa$, $\text{lcp}(1, 2) = 1 = |n|$, and $\text{lcp}(1, 5) = 4 = |\$\$\|n\$|$.

We classify a character in $T$ to be S-type and L-type and the last sentinel character $\$ is always S-type. $T[i]$ is said to be S- or L-type if $T[i] < T[i+1]$ or $T[i] > T[i+1]$, respectively. If $T[i] = T[i+1]$, $T[i]$ is said to be S- or L-type if $T[i+1]$ is S- or L-type, respectively. Correspondingly, a suffix $\text{suffix}(T, i)$ is said to be S- or L-type if $T[i]$ is S- or L-type. A suffix $\text{suffix}(T, i)$ is leftmost S-type (LMS) if $\text{suffix}(T, i)$ is S-type and $\text{suffix}(T, i-1)$ is L-type. The last suffix $\text{suffix}(T, n)$ is always LMS. An S-type suffix is also called S-suffix for short, so are L- and LMS-suffix.

The suffix array [24] $\text{SA}$ of a string $T$ is an array of length $n$, which contains a permutation of the integers $1..n$ such that $|\text{SA}[1..i]| < |\text{SA}[1..j]| < \ldots < |\text{SA}[1..n]|$. That is, for any $1 \leq i < j \leq n$, $\text{suffix}(T, \text{SA}[i]) < \text{suffix}(T, \text{SA}[j])$.

The inverse of $\text{SA}$ is denoted by $\text{ISA}$, which is another array of length $n$, which is the inverse permutation of $\text{SA}$ such that $\text{ISA}[\text{SA}[i]] = i$. $\text{ISA}[i] = j$ means that $\text{suffix}(T, i)$ is at the $j$-th position in $\text{SA}$. $\text{SA}$ or $\text{ISA}$ is an array only storing all the sorted L-, S- or LMS-suffixes.

The $\Phi$ array is defined by $\Phi[i] = \text{ISA}[\text{ISA}[i] - 1]$, which means that $\Phi[i]$ is the immediate lexicographical predecessor of the suffix $\text{suffix}(T, i)$. For completeness and computational convenience, we define $\Phi[\text{ISA}[1]] = 0$ and $\Phi[0] = \text{ISA}[n]$. Thus, $\Phi$ array is a cycle of length $n+1$.

The range of suffixes with an identical heading character $c$ is called a bucket in $\text{SA}$, denoted by $\text{bucket}(c)$. The number of buckets is equal to the size of $\Sigma$. Based on the concept of $\text{SA}$, it is easy to know that all buckets in $\text{SA}$ are sorted from the smallest to the largest. In each bucket, L-suffixes always followed by S-suffixes if both L- and S-suffixes exist. Hence, $\text{bucket}(c)$ can be further classified into at most two sub-buckets $\text{bucket}_L(c)$ and $\text{bucket}_S(c)$ for the L-suffixes and the S-suffixes, respectively. Given $T = \$\$abb\$$bbbbaa\$, Table 1 shows the $\text{SA}$, $\Phi$ and buckets of $\text{SA}$. There are two suffixes (10, 9) in $\text{bucket}(\$)$ and four suffixes (8, 7, 3, 4) in $\text{bucket}(a)$.

| $T$   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|
| $\text{SA}$ | (11) | (10) | 9  | 8  | 7  | 3  | 4  | 6  | 2  | 5  | 1  |    |
| $\text{bucket}(c)$ | $\$ | a  | b  | n  |     |    |    |    |    |    |    |    |
| $\Phi$   | 1  | 5  | 6  | 7  | 3  | 2  | 4  | 8  | 9  | 10 | 11 | 0  |

2.3 LZ77

The LZ77 factorization of a string $T$ is a left-to-right, greedy process that parses the string $T$ into the longest previous factors (LPF). The LPF at position $i$ is a pair $(p_i, l_i)$, where $p_i = T[i]$ and $l_i = 0$ if $T[i]$ does not occur before $i$. Otherwise, $p_i$ and $l_i$ is the position and length of the longest prefix of $\text{suffix}(T, i)$ that occurs at least once before $i$ in $T$, respectively. That is $T[i..i + l_i - 1] = T[p_i..p_i + l_i - 1]$ and $l_i > 0$ is maximized. For the example string $T = \$\$abb\$bb\$baa$, the LZ77 factorization produces:

$$(n, 0), (1, 1), (b, 0), (3, 1), (1, 4), (a, 0), (9, 1)$$

2.4 PSV and NSV

Crochemore and Ilie [10] showed that $p_i$ can be reduced to only two positions, the previous smaller values (PSVs) and the next smaller values (NSVs), which are defined as

$$\text{PSV(\text{SA}[i])] = \text{SA}[j_1]$$
$$\text{NSV(\text{SA}[i])] = \text{SA}[j_2]$$

where $j_1 = \max[j \in [1..i - 1]|\text{SA}[j] < \text{SA}[i])$ and $j_2 = \min[j \in [i + 1..n]|\text{SA}[j] < \text{SA}[i])$. If $j_1$ (or $j_2$) does not exist,
we set \( j_1 \) (or \( j_2 \)) equal to 0.

3. Previous Algorithms

3.1 KKP

KKP algorithm, which was proposed by Kärkkäinen et al. [12], has three versions: KKP3, KKP2 and KKP1. KKP3 computes the LZ77 of a string in two steps: the preliminary and the parsing step. In the preliminary step, the NSVs and PSVs for all positions are computed just by scanning the SA of \( T \) and stored in integer arrays in \( O(n) \) time, which uses the technique of the BGS algorithm by Goto and Bannai [13] (originated from [10], [25]). In the parsing step, all LZ factors can be sequentially acquired by NSVs and PSVs in \( O(n) \) time. During these two steps, KKP3 always needs to keep three integer arrays (SA, NSV and PSV) of length \( n \) in the preliminary step and two integer arrays (NSV and PSV) in the parsing step. Therefore the time complexity of KKP3 is \( O(n) \) and the working space of which is three integer arrays of length \( n \). KKP3 is closely related to the algorithm \( LZ, BG \) by Goto and Bannai [13]. However, it is more cache friendly by using some tricks while processing the NSVs and PSVs, and saves more space and time since it does not need auxiliary arrays of length \( n \) such as ISA or \( \Phi \) used in \( LZ, BG \).

KKP2 only computes NSVs in the preliminary step, and computes PSVs by NSVs in the parsing step. This is mainly due to the connection between \( PSV, NSV \) and \( \Phi \) arrays. During computing \( PSVs \), KKP2 rewrites the \( NSVs \) array in place such that one integer array of length \( n \) space can be saved. Therefore, \( PSVs \) and \( NSVs \) for all positions in string \( T \) can be sequentially acquired in \( O(n) \) time as well. In the whole computing process, KKP2 uses two integer arrays (SA and NSV) of length \( n \).

KKP1 is mainly based on KKP2 which has to keep two integer arrays of length \( n \) (SA and NSV). Since the values of SA are scanned sequentially and used only once, KKP1 stores the SA in the disk in the first, and then streams it from the disk while computing NSVs. Thus, there is only one integer array (NSV) of length \( n \) kept in memory. However, the total working space is still two integer arrays of length \( n \) (SA and NSV).

3.2 BGone

We describe the LZ factorization algorithm BGone proposed by Goto and Bannai [18]. BGone mainly focuses on how to compute LZ77 using only one integer array and in linear time, which uses the technique of induced sorting and simulates the SACA-K [26] algorithm for sorting L- and S-suffixes. It computes the LZ77 by three main steps.

1. Computing the \( \Phi \) array by induced sorting. In this step, all suffixes are linked from the largest one to the smallest one.

2. Computing the NSVs in-place from \( \Phi \).

3. Computing PSVs from NSVs in-place and then computes LZ77.

BGone is space economical. For in all these steps, BGone uses only one integer array of length \( n \) plus \( O(1) \) working space and runs in linear-time.

3.3 LZZone

BGone can compute the LZ77 in \( O(n) \) time. but the speed of BGone is a bit slow. So, our previous algorithm called LZZone [27] was designed for accelerating the speed of computing the LZ77, which has the same time and space complexities as BGone. LZZone has two computation paths for computing the LZ77 as follows:

1. If the L-suffixes outnumber the S-suffixes, all the suffixes are linked from the lexicographically smallest one to the largest one to form a \( \Psi \) array. Then, the PSV array can be computed by rewriting the \( \Psi \) in-place, and then the LZ77 is computed from PSV.

2. If the S-suffixes outnumber the L-suffixes, all the suffixes are linked from the lexicographically largest one to the smallest one to form a \( \Phi \) array. Then, the NSV array can be computed by rewriting the \( \Phi \) in-place, and then the LZ77 is computed from NSV.

For each computation path, LZZone just scans the working array for several passes in linear time and uses only one integer array during the whole computation process. So the time complexity for LZZone is \( O(n) \) and LZZone uses only \( 4n + O(1) \) bytes of working space. There are two versions of LZZone, i.e., LZZoneT and LZZoneSA, which compute the LZ77 from \( T \) and the SA of \( T \), respectively.

Figure 1 shows the differences of these algorithms. In LZZone, the LZ77 factorization is computed either from \( \Psi \) or \( \Phi \), it is unnecessary to compute \( \Psi_{inx} \). BGone, Xone and LZZone can compute the LZ77 factorization either from the SA of \( T \) or directly \( T \), while KKP and BGtwo must firstly compute the SA of \( T \). The computation path of Xone is the same as BGone. The key difference between Xone and BGone resides in how to compute \( \Psi_{inx} \) (see Sect. 4.2).
3.4 Rel-lz77

In order to compute LZ77 using small working space, Policriti and Prezza [20] proposed two algorithms: rel-lz77-1 and rel-lz77-2. Both rel-lz77-1 and rel-lz77-2 compute LZ77 in $O(r \log n)$ bits of working space and $O(n \log r)$ time, where $r$ is the number of runs in Burrows-Wheeler transform of $T$. rel-lz77-1 computes lz77 in two steps:

In the first step, the algorithm reads $T$ from left to right for building RLBWT.

In the second step, the algorithm scans $T$ from left to right and computes the LZ77 factors based on the RLBWT built in the first step.

The overall working space of rel-lz77-1 is lower bounded by about $3r$ words even with a careful implementation. By some observations that the size $z$ of the LZ77 parsing is often smaller than the number $r$ of runs in the Burrows-Wheeler transform in practice, and employing a suffix array sampling based on LZ77 phrases is better than on BWT runs, another space efficient solution rel-lz77-2 was proposed and the overall working space was reduced to about $r + 2z$ words. From the results of their experiments, the working space can be as small as 1% of the data set size in some cases.

4. This Work

Although LZone has achieved good speed improvement, LZone needs define more concepts such as LML-suffix and $SA_{lms}$. Moreover, LZone needs consider the number of L-suffixes and S-suffixes. Accordingly, two computation paths are required which increases the complexity of the program. We expect to make the algorithm more simpler and easier to implement in practice. So, we revisited the problem and found that the algorithm can run at a speed similar to that of LZone by redesigning the computing step with some features of suffix array. As a result, we come up this work. We named the algorithm Xone which also has two versions XoneT and XoneSA. Similar to BGoneT and BGoneSA, XoneT computes the LZ77 directly from $T$ while XoneSA computes the LZ77 from $SA$. The time complexity of Xone is $O(n)$ and the space requirement is $n \log n + O(\sigma \log n)$ bits.

We mainly describe XoneT. XoneSA is similar to XoneT except the method for getting $SA_{lms}$. The main difference between XoneT and BGoneT is how to compute $\Psi_{lms}$ from $SA_{lms}$. The computation steps of XoneT are shown as follows.

1. Computing $SA_{lms}$ from $T$
2. Computing $\Psi_{lms}$ from $SA_{lms}$
3. Computing $\Phi_r$ from $\Psi_{lms}$
4. Computing $\Phi$ from $\Phi_r$
5. Computing $NSV$ in-place from $\Phi$

6. Computing the LZ77 from $NSV$

4.1 Computing $SA_{lms}$

In order to compute the sorted $SA_{lms}$, both XoneT and BGoneT use the SACA-K algorithm by Nong [26]. SACA-K is space economical and based on induced sorting, which can be used to sort suffixes in linear time. The main idea of induced sorting algorithm is to sort a certain subset of suffixes directly or recursively, and then induce the lexicographic order of the remaining suffixes by the sorted subset suffixes.

As described in previous sections, the suffixes in $SA$ array are classified into two types: L- and S-suffixes. If all L- and S-suffixes are sorted, the $SA$ of $T$ is gotten. Based on Lemma 1 in [28], SACA-K firstly sorts the LMS-suffixes, then sorts the L-suffixes, finally sorts the S-suffixes.

**Lemma 1:** Given that all of the LMS-suffixes of $T$ are sorted, all of the suffixes of $T$ can be sorted in $O(n)$ time.

In order to sort the LMS-suffixes, SACA-K algorithm first defines LMS-substring and then performs the following two procedures for sorting LMS-substrings and L/S-suffixes, respectively.

1. Reducing procedure. SACA-K sorts all LMS-substrings recursively by induced sorting. In each reducing step, SACA-K gets a new string. The length of new string is at most half that of old string. For example, SACA-K gets $T_1$ after the first recursion, the length of $T_1$ is $|T_1| \leq |T|/2$. If all characters in $T_1$ are unique, which means that the lexicographic order of all the LMS-suffixes of $T$ can be decided directly, and the recursion ends. Otherwise, the recursion will continue until all characters in the new string $T_n$ are unique.

2. Inducing procedure. SACA-K performs inducing $SA_{i−1}$ from $SA_i$ for $n$ times, where $i$ decreases from $n$ to 1.

That is the inducing times is equal to that of reducing times. In each inducing step, two scans need to be performed: one is to scan sorted LMS-substrings for sorting L-suffixes, the other is to scan all sorted L-suffixes for sorting all S-suffixes. Thus, all suffixes are sorted and the $SA_i$ of string $T_i$ is gotten. When $i = 1$, $SA_1$ are achieved. That is all LMS-suffixes are sorted in working array $A[1..k]$, where $k$ is the number of LMS-suffixes. The more details about computing the $SA$ from the sorted LMS-suffixes by induced sorting see [26], [28].

It is clear that we need only the second-to-last inducing result, i.e., $SA_1$, which is just the sorted LMS-suffixes of string $T$. So, we just remove the last two steps of SACA-K for $SA_{lms}$. In the whole process, SACA-K uses only one integer array of length $n$ plus $O(\sigma \log n)$ bits of extra working space.
4.2 Computing $\Psi_{lms}$ from $SA_{lms}$

In order to sort all L-suffixes, we need scan all LMS-suffixes from the smallest one to the largest one. Since the working space to only one integer array of length $n$, we should turn $SA_{lms}$ into $\Psi_{lms}$ such that $A[0] = A[SA_{lms}[1]]$, $A[SA_{lms}[k]] = 0$, $A[i] = j$ if $suf(T, j)$ is the immediate lexicographical successor of $suf(T, i)$ in $SA_{lms}$. Thus all LMS-suffixes are linked and we can scan all LMS-suffixes in ascending order from $A[0]$.

Note that all LMS-suffixes are stored and stored in $A[1..k]$ before this step. If putting $suf(T, j)$ in $A[i]$, we may overwrite the LMS-suffix in $A[i]$ which has been used yet. We solve the problem as follows.

1. Define L- and S-position in working array $A$. The position in $A$ is an L- or S-position if $T[i]$ is an L or S-type character.

2. Put all LMS-suffixes in L-positions in $A$. Let $ptr = A[k]$, i.e., $ptr$ points to the largest LMS-suffix. Then, scan $T$ from $T[n]$ to $T[1]$, set $A[i] = ptr$ and $ptr = ptr - 1$ if $T[i]$ is L-type. By this way, all LMS-suffixes can be put in L-positions in $A$. There are enough L-positions in $A$ because there must be an L-type character in the left LMS-type character in $T$ according to the definition of LMS-suffix.

3. Link all LMS-suffixes from the smallest one to the largest one. Scan $A$ from left to right to visit all LMS-suffixes in L-positions and set $A[i] = A[j]$ if $suf(T, i)$ is the previous LMS-suffix of $suf(T, j)$ in $A$. The operation $A[i] = A[j]$ will not overwrite any LMS-suffixes because the type of position $i$ is S-type while the LMS-suffixes which has not been used are in L-positions. So, all LMS-suffixes can be linked properly.

This step requires to scan $A$ two times. One is to scan $A$ from right to left for putting LMS-suffixes in L-positions. The other is to link all LMS-suffixes. Therefore, this step runs in $O(n)$ time and uses only one integer array of length $n$.

In BGone, in order to compute $\Psi_{lms}$ and overcome the value in $A[i]$ be overwritten, the following steps are performed.

First, for all $1 \leq i \leq k$, set $A[2i] = A[i]$ and $A[2i - 1] = EMPTY$ by scanning $A[1,k]$ from right to left.

Next, for $1 \leq i \leq k - 1$, let $j_1 = A[2i]$, $j_2 = A[2i + 1]$, and set $A[j_1] = j_2$. In this case, the value in $A[j_1]$ may be overwritten for $A[j_1]$ stores some LMS-suffix. So, BGone borrow the space immediately left of position $j_1$ and set $A[j_1 - 1] = j_2$ if $A[j_1]$ is not empty. There is an important observation that LMS-suffixes cannot start at consecutive positions. That is, if $suf(T, j_1)$ is an LMS-suffix, $suf(T, j_1 - 1)$ must not be an LMS-suffix.

Finally, arranging the remaining values to there correct positions can be finished by traversing $SA_{lms}$ from $A[0]$ which stores the lexicographically smallest suffix of $SA_{lms}$.

If $suf(T, i)$ is the current traversing suffix, the succeeding suffix of $suf(T, i)$ must be stored in $A[i]$. If $A[i]$ is $EMPTY$, the succeeding suffix of $suf(T, i)$ must be stored in the immediately left position, i.e., $A[i - 1]$, then set $A[i] = A[i - 1]$. In this way, for each LMS-suffix $suf(T, i)$, the succeeding suffix of which can be stored in $A[i]$.

As can be seen from the above, BGone needs to scan the working array $A$ times while Xone scan $A$ times for getting $\Psi_{lms}$. So, Xone can save a lot of time in this step.

4.3 Computing $\Phi_T$ from $\Psi_{lms}$

In this step, we sort all L-suffixes from $\Psi_{lms}$ by induced sorting. Similar to the previous section, all L-suffixes should be sorted and linked from the largest one to the smallest one such that $A[0] = SA_{lms}[SA_{lms}[1]]$, $A[SA_{lms}[1]] = 0$, $A[i] = j$ if $suf(T, j)$ is the immediate lexicographical predecessor of $suf(T, i)$, i.e., to turn $\Psi_{lms}$ into $\Phi_T$. Thus, we can visit all L-suffixes in descending order from $A[0]$ for induced sorting S-suffixes in the next section.

From previous sections, we know that we should scan all buckets in $SA$ from the smallest one and to the largest one for sorting L-suffixes. When scanning a specific bucket $c$, we first scan $bucket_{c}[0]$ and then $bucket_{c}[k]$. Since all characters in $T$ come from the constant alphabet, the number of buckets is constant. We can enumerate all buckets for induced sorting L-suffixes. In previous section, all LMS-suffixes are linked from the smallest one to the largest one. Scanning LMS-suffixes of each $bucket_{c}(c)$ can be completed correctly. But scanning the L-suffixes and to put in its bucket seems difficult. To solve this problem and compute $\Phi$ in the next section, we simulate BGoneT[18] to scan a concrete bucket.

Firstly, define four integer arrays of length $\sigma$ for each character $c$: $bucket_{c}[0]$, $bucket_{c}[1]$, $bucke_{c}[2]$ and $bucket_{c}[3]$. $bucket_{c}[0]$ and $bucket_{c}[3]$ store the current start and end L-suffix of $bucket_{c}(c)$, respectively. $bucket_{c}[2]$ and $bucket_{c}[3]$ store the current start and end S-suffix of $bucket_{c}(c)$, respectively. All values of these buckets are initially to be empty before induced sorting.

Secondly, scan $\Psi_{lms}$ and update all $bucke_{c}[0]$ and $bucke_{c}[1]$. For each scanned LMS-suffix say $suf(T, i)$, if $bucke_{c}[T[0]]$ is empty, which means $suf(T, i)$ is the start and temporary last suffix of $bucket_{c}(T[i])$. We set $bucke_{c}[T[0]] = i$ and $bucke_{c}[T[0]] = i$. If $bucke_{c}[T[0]]$ is not empty, we just update $bucke_{c}[T[0]] = i$.

Finally, insert L-suffixes into $A$. Do the following for each scanned suffix say $suf(T, i)$.

1. If $suf(T, j)$ is an L-suffix and $j = i - 1$, we first check whether $bucke_{c}[T[j]]$ is empty. If $bucke_{c}[T[j]]$ is empty, which means that $suf(T, j)$ is the start and temporary end suffix of $bucket_{c}(T[j])$, we set $bucke_{c}[T[j]] = j$ and $bucke_{c}[T[j]] = j$. If $bucke_{c}[T[j]]$ is not empty, which means that there must exist one suffix lexicographical smaller than $suf(T, j)$ in $bucket_{c}(T[j])$. The smaller suffix is
\( Lbu\text{ckete}[T[j]] \), which is also the immediate lexicographical predecessor of \( su\text{f}(T, j) \). If \( Lbu\text{ckete}[T[j]] = p \), we set \( A[p] = j \) and update \( Lbu\text{ckete}[T[j]] = j \). Thus \( su\text{f}(T, j) \) is the current end and largest suffix of \( bue\text{ckete}(c) \).

2. If \( su\text{f}(T, j) \) is an S-suffix, we do nothing.

In this way, all L-suffixes can be put in the appropriate positions in \( A \) and \( \Psi_{\text{sm}} \) is turned into \( \Phi_{l} \). In the whole process, scanning \( A \) is in linear time and reading or writing the values of \( A \) can be done in \( O(1) \) time. The working space requirement remains one integer array of length \( n \) plus \( O(1) \) extra working space.

4.4 Computing \( \Phi \) from \( \Phi_{l} \)

In this step, we should sort all S-suffixes from the sorted L-suffixes, i.e., turn \( \Phi_{l} \) into \( \Phi \) such that \( A[0] = SA[n] \), \( A[SA[1]] = 0 \). \( A[i] = j \) if \( su\text{f}(T, j) \) is the immediate lexicographical predecessor of \( su\text{f}(T, i) \). Thus, we can visit all suffixes in descending order from \( A[0] \) for computing NSVs in the next section.

In Sect. 4.3, all L-suffixes are linked from the smallest one to the largest one. Since induced sorting S-suffixes should scan all L-suffixes from the largest one to the smallest one, we should first reverse the direction of all links in \( A \) by \( A[i] = j \) if \( su\text{f}(T, j) \) is the lexicographical predecessor of \( su\text{f}(T, i) \). When scanning a specific \( bue\text{cket}(c) \), we first scan \( bue\text{ckete}(c) \) and then \( bue\text{cket}(c) \). We also define four integer arrays of length \( \sigma \) for induced sorting S-suffixes: \( Lbu\text{cketes}[c], Lbu\text{cketc}[c], Sbu\text{cketes}[c] \) and \( Sbu\text{cketc}[c] \), and perform the following steps for induced sorting S-suffixes.

Firstly, scan \( \Phi_{l} \) and update all \( Lbu\text{cketes}[c] \) and \( Lbu\text{cketc}[c] \) for recording the start and end suffix of each \( bue\text{cket}(c) \). For each scanned L-suffix say \( su\text{f}(T, i) \), if \( bue\text{ckete}(T[i]) \) is empty, which means \( su\text{f}(T, i) \) is the end and temporary start suffix of \( bue\text{cket}(c) \). We set \( Lbu\text{cketes}[c] = i \) and \( Lbu\text{cketc}[c] = i \).

Secondly, insert S-suffixes into \( A \). Scan all buckets from the largest one to the smallest one. During scanning process, all L- or S-suffixes will be visited. Do the following for each scanned suffix say \( su\text{f}(T, i) \).

1. If \( su\text{f}(T, j) \) is an S-suffix and \( j = i - 1 \), we first check whether \( bue\text{ckete}[T[j]] \) is empty. If \( bue\text{ckete}[T[j]] \) is empty, which means that \( su\text{f}(T, j) \) is the end and temporary start suffix of \( bue\text{cket}(c) \), we set \( bue\text{ckete}[T[j]] = j \) and \( bue\text{cketes}[T[j]] = j \). If \( bue\text{ckete}[T[j]] \) is not empty, which means that there must exist one suffix lexicographical larger than \( su\text{f}(T, j) \) in \( bue\text{ckete}[T[j]] \). The larger suffix is \( bue\text{cketes}[T[j]] \), which is also the immediate lexicographical successor of \( su\text{f}(T, j) \). If \( bue\text{cketes}[T[j]] = p \), we set \( A[p] = j \) and update \( bue\text{ckete}[T[j]] = j \). Thus \( su\text{f}(T, j) \) is the current start and smallest suffix of \( bue\text{ckete}(c) \).

2. If \( su\text{f}(T, j) \) is an L-suffix, we do nothing.

When all buckets have been scanned, all suffixes are linked from the largest one to the smallest one except the suffixes at boundaries between \( bue\text{cket}(c) \) and \( bue\text{cket}(c) \). Since \( Lbu\text{cketes}[c] \) and \( Lbu\text{ckete}[c] \) store the start and end L-suffix of \( bue\text{cket}(c) \), and \( Sbu\text{cketes}[c] \) and \( Sbu\text{ckete}[c] \) store the start and end S-suffix of \( bue\text{cket}(c) \), respectively. We just scan all these buckets in decreasing order to link suffixes at boundaries. Now, all suffixes are linked and the working array \( A \) becomes a \( \Phi \) array.

In this step, computing \( \Phi \) runs in linear time and the working space requirement remains only one integer array of length \( n \) plus \( O(\sigma \log n) \) bits extra working space.

4.5 Computing NSVs from \( \Phi \) In-Place

Goto and Bannai [18] showed that NSVs can be computed in-place from \( SA \) by constructing \( \Phi \) array and peak elimination [10]. For the completeness of presentation, we describe more details about these tricks here, which is also used in XoneT.

KKP2 computes NSVs by scanning \( SA \) from the right to the left sequentially. That is, computing NSVs requires scanning all suffixes of \( SA \) from the largest one to the smallest one. For each scanned suffix \( su\text{f}(T, i) \), NSVs[\( su\text{f}(T, i) \)] can be computed by peak elimination. The more details about peak elimination are described in [10]. For saving working space, Goto and Bannai compute NSVs from \( SA \) in-place. It seems difficult because the values of \( SA \) are in lexicographical order, while the values of NSVs are in text order. Goto and Bannai solve this problem by constructing the \( \Phi \) array. In previous sections, we know that \( \Phi[i] \) is the immediate lexicographical predecessor of \( su\text{f}(T, i) \) and \( \Phi[0] \) is the largest suffix of \( SA \). Starting from \( \Phi[0] \), all suffixes can be visited from the largest one to the smallest one. Thus computing NSVs from \( SA \) can be simulated. Since the access on \( SA \) is sequential and the value in \( \Phi[i] \) is not needed any more after it has been processed. \( \Phi[i] \) can be reused to store NSVs[i]. In this way, NSVs can be computed in-place and in \( O(n) \) time. The algorithm for computing NSVs from \( SA \) in-place is shown in Algorithm 1.
Algorithm 2: Computing pair \((p_i, l_i)\) at position \(i\) of string \(T\).

| Input: The string \(T\), \(nsv\) and \(psv\) |
| Output: \((p_i, l_i)\) |
| 1 compute_pair\((T, nsv, psv)\); |
| 2 if \(LCP(i, nsv) < LCP(i, psv)\) then |
| 3 \((p_i, l_i) \leftarrow (psv, LCP(i, psv))\); |
| 4 else |
| 5 \((p_i, l_i) \leftarrow (nsv, LCP(i, nsv))\); |
| 6 end |
| 7 if\(i = 0\) then \(p_0 = T[i]\); |
| 8 return \((p_i, l_i)\) |

4.6 Computing LZ77 from NSVs

If NSVs have been acquired, PSVs can be computed in-place in the parsing step by Lemma 2, which uses the relation between PSV, NSV and Φ arrays.

Lemma 2: (\cite{12}) Given the NSV array of a string \(T\) of length \(n\), \(PSV(i)\) and \(NSV(i)\) of \(T\) can be sequentially obtained for all positions \(i = 1, \ldots, n\) in \(O(n)\) total time using \(O(\log n)\) bits space other than the NSV array and \(T\).

The algorithm for computing pair at position \(i\) of string \(T\) can be described in Algorithm 2.

Now, both NSVs and PSVs have been computed, we can compute pair \((p_i, l_i)\) by comparing the values of \(LCP(i, NSV[i])\) and \(LCP(i, PSV[i])\) and choosing the larger one. All LZ factors can be sequentially computed just by scanning \(T\) from left to right. In this step, the time complexity is still \(O(n)\) and the working space requirement remains to one integer array of length \(n\).

5. Experimental Results

5.1 Setup

We performed experiment on a computer with 1 CPU (Intel(R) Xeon(R) 3.00 GH, E3-1220 v6, 4 cores, 4 threads), 8 GiB RAM (DDR4, 2,400 MHz). The operation system was Linux (U buntu 16.04.6, 64-bit). All programs were compiled using g++ version 5.2.1 with “-fomit-frame-pointer -W -Wall -Winline -NDNDBG -O3” options. All reported runtimes were measured in seconds and the average of 3 runs was reported.

5.2 Data Set

For experiments, we used the following datasets.\footnote{The first five datasets can be downloaded from http://pizzachili.dcc.uchile.cl/texts.html} Table 2 shows the more details of these datasets.

- proteins.200: this file is a sequence of newline-separated protein sequences (without descriptions, just the bare proteins) obtained from the Swissprot database. Each of the 20 amino acids is coded as one uppercase letter.
  - english.200: this file is the concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. We deleted the headers related to the project so as to leave just the real text.
  - dna.200: this file is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hg1p10 to 21hg1p10, plus 0xhp10 and 0yhp10, from Gutenberg Project. Each of the 4 bases is coded as an uppercase letter A, G, C, T, and there are a few occurrences of other special characters.
  - sources.200: this file is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6, 11.6 and gcc-4.0.0 distributions.
  - cere: this file is the baking yeast genome, which contains only five types of characters.
  - coreutils:\footnote{Available at http://www.gnu.org/software/coreutils/} this file is the GNU Core Utilities, which are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.
  - kernel:\footnote{Available at http://www.kernel.org} source files from Linux kernel.
  - einstein.en.txt:\footnote{Available at http://dumps.wikimedia.org/} this file is Wikipedia dumps in XML format concatenated.

5.3 Comparision of Algorithms

We compare the performance of Xone described in this paper to KKP3, BGtwo, Bgone, LZone, gzip (version 1.6) and rel-lz77-2, where gzip is a file compression utility software in Linux. XoneT, LZoneT and BGoneT compute LZ77 directly from \(T\), while XoneSA, LZoneSA and BGoneSA compute LZ77 from SA. The SA is computed by calling \texttt{divsufsort}\footnote{Available at https://code.google.com/p/libdivsufsort/}. The results are given in Table 3. The mean time for each algorithm is the total time divided by the total number of characters of all the input data.

The experimental results show that for computing the LZ77 from an input string, KKP3 and gzip are about 0.37/0.14 - 1 = 1.64 and 0.37/0.06 - 1 = 5.17 faster than XoneT, respectively. XoneT and XoneSA are about
Table 3
Times for computing LZ77.

| Algorithm   | KKP  | BGtwo | BGoneT | BGoneSA | XoneT | XoneSA | LZoneT | LZoneSA | gzip | rle-lz77-2 |
|-------------|------|-------|--------|---------|-------|--------|--------|---------|------|------------|
| proteins.200| 38.3 | 55.21 | 106.81 | 93.1    | 89.8  | 75.04  | 95.25  | 65.67   | 7.85 | 12374      |
| english.200  | 32.73| 51.08 | 105.99 | 93.1    | 89.8  | 75.04  | 95.25  | 65.67   | 7.85 | 12374      |
| dna.200      | 38.3 | 55.21 | 106.81 | 93.1    | 89.8  | 75.04  | 95.25  | 65.67   | 7.85 | 12374      |
| sources.200  | 21.66| 39.94 | 76.67  | 66.44   | 61.62 | 51.38  | 64.21  | 44.37   | 6.40 | 12046      |
| coreutils    | 21.4 | 40.07 | 75.7   | 67.45   | 58.7  | 49.9   | 61.81  | 42.89   | 6.34 | 10223      |
| cere         | 62.57| 113.49| 214.25 | 197.6   | 163.42| 147.74| 174.68 | 125.13  | 50.51| 18095      |
| kernel       | 27.1 | 51.66 | 100.5  | 88.04   | 78.3  | 65.37  | 81.95  | 57.72   | 8.89 | 12196      |
| einstein.en.txt| 57.63| 114.41| 223.4  | 187.65  | 176.48| 142.14| 187.9  | 119.45  | 21.19| 21246      |
| total (s)    | 295.28| 519.75| 1002.49| 884.38  | 792.77| 676.55| 842.54| 581.99  | 137.74| 109779     |
| mean (s/MiB) | 0.14 | 0.24  | 0.47   | 0.42    | 0.37  | 0.32   | 0.40   | 0.27    | 0.06 | 51.37      |

Fig. 2 The runtime of BGoneT and XoneT in different phrases. In each data set, the runtime of BGoneT and XoneT are shown above and below, respectively.

0.47/0.37 − 1 = 0.27 and 0.42/0.32 − 1 = 0.31 faster than BGoneT and BGoneSA, respectively. XoneT is about 0.40/0.37 − 1 = 0.08 faster than LZoneT, while LZoneSA runs about 0.32/0.27 − 1 = 0.18 faster than XoneSA. rle-lz77-2 runs the slowest, but it saves the most working space.

BGone, LZone and Xone run slower than KKP algorithms. One reason mainly due to that BGone, LZone and Xone algorithms need compute Φ/Ψ array before computing NSV/PSV, while KKP does not. Another reason could be that during the process of computing NSV or PSV, BGone, LZone and Xone algorithms need random access on the working array, while KKP reads the SA in a sequential way, which saves lots of time.

We also note that XoneT runs faster than LZoneT while XoneSA runs slower than LZoneSA. This is mainly due to that LZoneSA computes Φr directly from SR which can be acquired just by scanning SA, while XoneSA needs compute Ψlms before computing Φr. So, given SA, LZoneSA has more advantages. As for XoneT and LZoneT, XoneT gets SAlms by taking the intermediate result of SACA-K in the last inducing step. LZoneT also gets SR by the same way, but LZoneT needs one more step to scan SAlms for getting SR. In most instances, the length of SR is often longer than Ψlms, to scan SR will take more time. Although XoneT needs compute Ψlms before computing Ψlms, in the whole, XoneT saves more time than LZoneT.

To further compare the two algorithms, we divided the computation process into 3 phrases:
1. Compute SAlms from T;
2. Compute Φr from SAlms;
3. Compute LZ77 from Φr. Figure 2 shows the runtime in each phase. There is little difference between the two algorithms in the first and third phase in terms of speed. But in the second phase, XoneT is significantly faster than BGoneT. Figure 3 shows the runtime of XoneSA and BGoneSA for computing LZ77 from SA. The result shows that given SA, XoneSA runs much faster than BGoneSA.

6. Conclusions

We showed a linear-time algorithm called Xone for computing LZ77, which uses only n log n + O(σ log n) bits of working space. Although Xone runs slower than KKP, Xone is more space economical and provides more choice for computing LZ77. In the whole computing process, both BGone and Xone uses a number of techniques to rewrite the various auxiliary integer arrays from one to another in-place and in linear time, which saves a lot of working space. The key difference between BGone and Xone is that how to compute Ψlms from SAlms. For computing Ψlms, BGone takes advantage of the feature that two LMS-suffixes cannot be start at consecutive positions for avoiding overwriting. Thus BGone needs to scan the working array A times. While

Fig. 3 Given SA, the runtime of BGoneSA and XoneSA.
Xone defines the L- and S-positions in SA, and put the LMS-suffixes into L-positions for avoiding overwriting, which requires to scan A only twice, so better efficiency is obtained. BGone, LZone and Xone needs compute auxiliary array such as Φ or Ψ array, and then compute NSV or PSV array. All these computations involve a large number of random access on the working array. Random access seems slower than sequential access. In future work, random access should be avoided as far as possible.

From Fig. 2, XoneT gives a faster way for compute Ψ_{ims}, while there is still no significant improvement in the time of computing Ψ_{ims} from T. We leave open the problem whether this can be further accelerated. One solution could be use parallelization to compute Ψ_{ims} and use compressed suffix array for further space saving.

References

[1] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. Inf. Theory, vol.23, no.3, pp.337–343, May 1977.
[2] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S.J. Puglisi, “A faster grammar-based self-index,” in Language and Automata Theory and Applications, ed. A.H. Dediu and C. Martin-Vide, Lecture Notes in Computer Science, vol.7183, pp.240–251, Springer-Verlag, Berlin, Germany, 2012.
[3] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S.J. Puglisi, “LZ77-based self-indexing with faster pattern matching,” in Combinatorial Pattern Matching, ed. A. Pardo and A. Viola, Lecture Notes in Computer Science, vol.8392, pp.731–742, Springer-Verlag, Berlin, Germany, 2014.
[4] G. Navarro and N. Prezza, “Universal compressed text indexing,” Theoretical Computer Science, vol.762, pp.41–50, 2019.
[5] W. Thies and S. Hall, “Manipulating lossless video in the compressed domain,” Proc. 17th ACM international conference on Multimedia, MM ’09, New York, NY, USA, pp.331–340, ACM, 2009.
[6] J. Abell, “Incremental frequency count - a post BWT-stage for the Burrows-Wheeler compression algorithm,” Software: Practice and Experience, vol.37, no.3, pp.247–265, March 2007.
[7] A. Langu, “On parsing optimality for dictionary-based text compression - the Zip case,” Journal of Discrete Algorithms, vol.20, no.0, pp.65–70, May 2013.
[8] J. Fang, J. Chen, Z. Al-Ars, H.P. Hofstee, and J. Lee, “An efficient high-throughput LZ77-based decompressor in reconfigurable logic,” Journal of Signal Processing Systems, pp.1–17, 2020.
[9] G. Chen, S.J. Puglisi, and W.F. Smyth, “Fast and practical algorithms for computing all the runs in a string,” in Combinatorial Pattern Matching, ed. B. Ma and K. Zhang, Lecture Notes in Computer Science, vol.4580, pp.307–315, Springer-Verlag, Berlin, Germany, 2007.
[10] M. Crochemore and L. Ilie, “Computing longest previous factor in linear time and applications,” Information Processing Letters, vol.106, no.2, pp.75–80, April 2008.
[11] J. Kärkkäinen and P. Sanders, “Simple linear work suffix array construction,” in Automata, Languages and Programming, ed. J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, Lecture Notes in Computer Science, vol.2719, pp.943–955, Springer-Verlag, Berlin, Germany, 2003.
[12] J. Kärkkäinen, D. Kempa, and S.J. Puglisi, “Linear time Lempel-Ziv factorization: Simple, Fast, Small,” in Combinatorial Pattern Matching, ed. J. Fischer and P. Sanders, Lecture Notes in Computer Science, vol.7922, pp.189–200, Springer-Verlag, Berlin, Germany, 2013.
[13] K. Goto and H. Bannai, “Simpler and faster Lempel Ziv factorization,” Proc. 2013 Data Compression Conference, DCC ’13, Washington, USA, pp.133–142, IEEE Computer Society, 2013.
[14] A. Policriti and N. Prezza, “Fast online Lempel-Ziv factorization in compressed space,” String Processing and Information Retrieval, ed. C. Iliopoulos, S. Puglisi, and E. Yilmaz, Cham, pp.13–20, Springer International Publishing, 2015.
[15] T. Nishimoto and Y. Tabei, “LZRR: LZ77 parsing with right reference,” 2019 Data Compression Conference (DCC), pp.211–220, 2019.
[16] D. Kosolobov, D. Valenzuela, G. Navarro, and S.J. Puglisi, “Lempel-Ziv-like parsing in small space,” Algorithmica, vol.82, no.1, pp.3195–3215, 2020.
[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, “Linear-time longest-common-prefix computation in suffix arrays and its applications,” Combinatorial Pattern Matching, ed. A. Amir, Berlin, Heidelberg, pp.181–192, Springer Berlin Heidelberg, 2001.
[18] K. Goto and H. Bannai, “Space efficient linear time Lempel-Ziv factorization for small alphabets,” Proc. 2014 Data Compression Conference, DCC ’14, Washington, USA, pp.163–172, IEEE Computer Society, 2014.
[19] A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylova, W.F. Smyth, G. Tischler, and M. Yusufu, “A comparison of index-based Lempel-Ziv LZ77 factorization algorithms,” ACM Computing Surveys, vol.45, no.1, pp.5.1–5.17, 2013.
[20] A. Policriti and N. Prezza, “LZ77 computation based on the run-length encoded BWT,” Algorithmica, no.7, pp.1–26, 2018.
[21] A. Cramaj, P. Davies, and M. Parter, “Component stability in low-space massively parallel computation,” Proc. 2021 ACM Symposium on Principles of Distributed Computing, PODC’21, New York, NY, USA, pp.481–491, Association for Computing Machinery, 2021.
[22] D. Kempa and B. Langmead, “Fast and space-efficient construction of AVL grammars from the LZ77 parsing,” Proc. 2021 European Symposium on Algorithms, ESA 2021, Lisbon, Portugal, pp.56:1–56:14, 2021.
[23] Y. Liu, T. Hankeln, and B. Schmidt, “Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data,” IEEE/ACM Trans. Comput. Biol. Bioinform., vol.13, no.3, pp.592–598, 2016.
[24] U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” SIAM Journal on Computing, vol.22, no.5, pp.935–948, 1993.
[25] E. Ohlebusch and S. Gog, “Lempel-Ziv factorization revisited,” Combinatorial Pattern Matching, ed. R. Giancarlo and G. Manzini, Berlin, Heidelberg, pp.15–26, Springer Berlin Heidelberg, 2011.
[26] G. Nong, “Practical linear-time O(1)-workspace suffix sorting for constant alphabets,” ACM Transactions on Information Systems (TOIS), vol.31, no.3, July 2013.
[27] W.J. Liu, G. Nong, W.H. Chan, and Y. Wu, “Improving a lightweight LZ77 computation algorithm for running faster,” Software: Practice and Experience, vol.46, no.9, pp.1201–1217, 2016.
[28] G. Nong, S. Zhang, and W.H. Chan, “Two efficient algorithms for linear time suffix array construction,” IEEE Trans. Comput., vol.60, no.10, pp.1471–1484, Oct. 2011.