Vol.2013-AL-145 No.6
2013/11/6

高速かつ省領域な線形時間LZ分解アルゴリズム

Keisuke Goto[1,2,a)] Hideo Bannai[1,b)]

Abstract: We present a new algorithm for computing the Lempel-Ziv Factorization (LZ77) of a given string of length N in linear time, that utilizes only $N \log N + O(1)$ bits of working space, i.e., a single integer array, for constant size integer alphabets. This greatly improves the previous best space requirement for linear time LZ77 factorization (Kärkkäinen et al. CPM 2013), which requires two integer arrays of length N. Computational experiments show that despite the added complexity of the algorithm, the speed of the algorithm is only around twice as slow as previous fastest linear time algorithms.

1. Introduction

Lempel-Ziv (LZ77) factorization [16] is one of the most important concepts in string processing with countless applications in compression [15], [16], as well as efficient string processing [4], [10]. More recently, its importance has been reasserted due to the highly repetitive characteristics of modern datasets, such as collections of genome sequences, for which compression schemes based on LZ77 have been shown to be particularly effective [11]. Thus, time and space efficient computation of LZ77 factorization is a very important and heavily studied topic (See [1] for a survey).

In this paper, we focus on worst case linear time algorithms for computing the LZ77 factorization of a given text. All existing linear time algorithms are based on the suffix array, which can be constructed in linear time independent of alphabet size, when assuming an integer alphabet. The earlier algorithms further compute and utilize several other auxiliary integer arrays of length N, such as the inverse suffix array, the longest common prefix (LCP) array [9], and the Longest Previous Factor (LPF) array [3], and thus until recently, required at least 3 auxiliary integer arrays of length N in addition to the text. Since all values in the LCP and LPF arrays are not required for computing the LZ factorization, the most efficient recent linear time algorithms [5], [6] avoid constructing these arrays altogether.

The currently fastest linear time LZ-factorization algorithm, as well as the currently most space economical linear time LZ-factorization algorithm, have been proposed by Kärkkäinen et al. [6] They proposed 3 algorithms KKP3, KKP2, and KKP1, which respectively store and utilize 3, 2, and 1 auxiliary integer arrays of length N kept in main memory. All three algorithms compute the LZ-factorization of the input text given the text and its suffix array. KKP3 is very similar to LZ_BG [5], but is modified so that array accesss are more cache friendly, thus making the algorithm run faster. KKP2 is based on KKP3, but further reduces one integer array by an elegant technique that rewrites values on the integer array. KKP1 is the same as KKP2, except that it assumes that the suffix array is stored on disk, but since the values of the suffix array are only accessed sequentially, the suffix array is streamed from the disk. Thus, KKP1 can be regarded as requiring only a single integer array to be held in memory. In this sense, KKP1 is the most space economical linear time algorithm, and has been shown to be faster than KKP2, if we assume that the suffix array is already computed and exists on disk [6]. However, note that the *total* space requirement of KKP1 is still two integer arrays, one existing in memory and the other existing on disk.

In this paper, we propose new algorithms for computing the LZ77 factorization that uses only a single auxiliary integer array of length N. We achieve this by introducing a series of techniques for rewriting the various auxiliary integer arrays from one to another, in linear time and in-place, i.e., using only constant extra space. Computational experiments show that our algorithm is at most around twice as slow as previous algorithms, but in turn, uses only half the total space, and may be a viable alternative when the total space (including disk) is a limiting factor due to the enormous size of data.

2. Preliminaries

Let Σ be a finite *alphabet*. In this paper, we assume that Σ is an integer alphabet of constant size. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The set of prefixes of T is denoted by $\mathit{prefix}(T)$. The *longest common prefix* of strings X, Y, denoted $\mathit{lcp}(X, Y)$, is the longest string in $\mathit{prefix}(X) \cap \mathit{prefix}(Y)$.

The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i..j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i..j] = \varepsilon$ if $j < i$, $\mathit{suf}(i)$ indicates $T[i..|T+1|]$,

1 Department of Informatics, Kyushu University, Japan
2 Japan Society for the Promotion of Science (JSPS)
a) keisuke.gotou@inf.kyushu-u.ac.jp
b) bannai@inf.kyushu-u.ac.jp

© 2013 Information Processing Society of Japan

and $T[|T|+1] = \$$ where \$ is a special delimiter character that does not occur elsewhere in the string.

2.1 Suffix Arrays

The suffix array [12] SA of any string T is an array of length $|T|$ such that for any $1 \le i \le |T|$, $SA[i] = j$ indicates that $suf(j)$ is the i-th lexicographically smallest suffix of T. For convenience, we assume that $SA[0] = SA[N+1] = 0$. The inverse suffix array SA^{-1} of SA is an array of length $|T|$ such that $SA^{-1}[SA[i]] = i$. As in [7], let Φ be an array of length $|T|$ such that $\Phi[SA[1]] = |T|$ and $\Phi[SA[i]] = SA[i-1]$ for $2 \le i \le |T|$, i.e., for any suffix $j = SA[i]$, $\Phi[j] = SA[i-1]$ is the immediately preceding suffix in the suffix array. The suffix array SA for any string of length $|T|$ can be constructed in $O(|T|)$ time regardless of the alphabet size, assuming an integer alphabet (e.g. [8], [14]). Furthermore, there exists a linear time suffix array construction algorithm for a constant alphabet using $O(1)$ working space [13].

2.2 LZ Encodings

LZ encodings are dynamic dictionary based encodings with many variants. The variant we consider is also known as the s-factorization [2].

Definition 1 (LZ77-factorization)**.** *The s-factorization of a string T is the factorization $T = f_1 \cdots f_n$ where each s-factor $f_k \in \Sigma^+$ $(k = 1, \ldots, n)$ starting at position $i = |f_1 \cdots f_{k-1}| + 1$ in T is defined as follows: If $T[i] = c \in \Sigma$ does not occur before i then $f_k = c$. Otherwise, f_k is the longest prefix of $suf(i)$ that occurs at least once before i.*

Note that each LZ factor can be represented in constant space, i.e., a pair of integers where the first and second elements respectively represent the length and position of a previous occurrence of the factor. If the factor is a new character and the length of its previous occurrence is 0, the second element will encode the new character instead of the position. For example the s-factorization of the string $T =$ `abaababababaaaaabbabab` is `a`, `b`, `a`, `aba`, `baba`, `aaaa`, `b`, `babab`. This can be represented as $(0, \mathtt{a})$, $(0, \mathtt{b})$, $(1, 1)$, $(3, 1)$, $(4, 5)$, $(4, 10)$, $(1, 2)$, $(5, 5)$.

We define two functions LPF and $PrevOcc$ below. For any $1 \le i \le N$, $LPF(i)$ is the longest length of longest common prefix between $suf(i)$ and $suf(j)$ for any $1 \le j < i$, and $PrevOcc(i)$ is a position j which gives $LPF(i)$[*1]. More precisely,

$$LPF(i) = \max(\{0\} \cup \{lcp(suf(i), suf(j)) \mid 1 \le j < i\})$$

and

$$PrevOcc(i) = \begin{cases} -1 & \text{if } LPF(i) = 0 \\ j & \text{otherwise} \end{cases}$$

where j satisfies $1 \le j < i$, and $T[i : i + LPF(i) - 1] = T[j : j + LPF(i) - 1]$. Let $p_k = |f_1 \cdots f_{k-1}| + 1$. Then, f_k can be represented as a pair $(LPF(p_k), PrevOcc(p_k))$ if $LPF(p_k) > 0$, and $(0, T[p_k])$ otherwise.

Crochemore and Ilie [3] showed that candidates values for $PrevOcc(i)$ can be reduced to only 2 position, namely, the previous smaller value (PSV) and the next smaller value (NSV) [3], which are defined as follows:

$$PSV[i] = SA[j_1]$$
$$NSV[i] = SA[j_2]$$

where $j_1 = \max(\{0\} \cup \{1 \le j < SA^{-1}[i] \mid SA[j] < SA[i]\})$ and $j_2 = \min(\{N+1\} \cup \{N \ge j > SA^{-1}[i] \mid SA[j] < SA[i]\})$.

In what follows, we assume that the algorithms output each LZ factor sequentially, and will not include the total size of the LZ factorization in the working space.

3. Previous Algorithm

We first describe the 3 variants (KKP3, KKP2, and KKP1) of the LZ factorization algorithm proposed by Kärkkäinen et.al [6].

KKP3 consists of two steps, which we shall call the preliminary step and the parsing step. In the preliminary step, KKP3 computes PSV and NSV for all positions and stores them in integer arrays. Although we defer the details, the PSV and NSV arrays can be computed in linear time by sequentially scanning SA of T, and is based on the peak elimination by Crochemore and Ilie [3]. Then, in the parsing step, KKP3 computes the LZ-factorization by a naive comparison between $suf(i)$ and $suf(PSV[i])$, as well as $suf(i)$ and $suf(NSV[i])$, for all positions i that a factor starts (See Algorithm 1. $lcp(i, j)$ computes the length of the longest prefix between $suf(i)$ and $suf(j)$ in $O(lcp(i, j))$ time). In order to compute a factor f_j, the algorithm compares at most twice $|f_j|$ characters. Since the sum of the length of all the factors is N, the parsing step of the algorithm runs in linear time. KKP3 needs 3 integer arrays, SA, PSV and NSV arrays in the preliminary step, and 2 integer arrays PSV and NSV in the parsing step. Therefore KKP3 runs in linear time using a total of 3 auxiliary integer arrays (SA, PSV, NSV) of length N.

Algorithm 1: Computing the LZ77 factorization from SA via PSV and NSV arrays (KKP3)

Input : Suffix Array $SA[1..N]$ of string T of length N
1 $SA[0] \leftarrow 0$;
2 $SA[N+1] \leftarrow 0$;
3 **for** $i \leftarrow 1$ **to** $N+1$ **do**
4   **while** $SA[top] > SA[i]$ **do**
5     $NSV[SA[top]] \leftarrow SA[i]$;
6     $PSV[SA[top]] \leftarrow SA[top-1]$;
7     $top \leftarrow top - 1$;
8   $top \leftarrow top + 1$;
9   $SA[top] \leftarrow SA[i]$;
10 **while** $i \le n$ **do**
11   $lcp_{nsv} \leftarrow lcp(i, NSV[i])$; `// return 0 if` $NSV[i] = 0$
12   $lcp_{psv} \leftarrow lcp(i, PSV[i])$; `// return 0 if` $PSV[i] = 0$
13   $l \leftarrow -1$;
14   $p \leftarrow T[i]$;
15   **if** $lcp_{nsv} > 0$ **and** $lcp_{nsv} \ge lcp_{psv}$ **then** $l \leftarrow lcp_{nsv}$; $p \leftarrow NSV[i]$;
16   **else if** $lcp_{psv} > 0$ **then** $l \leftarrow lcp_{psv}$; $p \leftarrow PSV[i]$;
    Output: $((l, p))$
17   $i \leftarrow i + \max(1, l)$;

For KKP2, Kärkkäinen et al. show that the parsing step can be accomplished by using only the NSV array. The idea is based on a very interesting connection between PSV, NSV, and Φ arrays.

[*1] There can be multiple choices of j, but here, it suffices to fix one.

© 2013 Information Processing Society of Japan

They showed that starting from the *NSV* array, it is possible to sequentially scan and rewrite the *NSV* array (consequently to the Φ array) in-place, during which, values of *PSV* (and naturally *NSV*) for each position can be obtained sequentially as well.

Lemma 1 ([6]). *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 $O(1)$ space other than the NSV array and T.*

By making use of this technique, only the *NSV* array is now required for the parsing step. KKP2 uses 2 integer arrays (*SA* and *NSV*) in the preliminary step, and 1 integer array (*NSV*) in the parsing step, and thus in summary, KKP2 runs in linear time using a total of 2 auxiliary integer arrays of length N.

We can see that the memory bottleneck of KKP2 is in the preliminary step, i.e., the computation of the *NSV* array, where the space for *SA* is required as input, and the space for *NSV* is required for output. This is because elements of *SA* are in lexicographic order and elements of *NSV* are in text order. Although the scanning on *SA* can be sequential, the writing to *NSV* is not, and both arrays must exist simultaneously. KKP1 partly overcomes this problem, by first storing *SA* to disk, and then streams the *SA* from the disk, storing only the *NSV* array in main memory. Thus, KKP1 runs in linear time keeping only 1 auxiliary integer array of length N in *main* memory, although of course, the total storage requirement is still 2 integer arrays (*SA* and *NSV*).

4. New Algorithm using a single integer array

In this section, we describe our linear time LZ77 factorization algorithm that uses only a single auxiliary integer array of length N. As described in the previous section, once the *NSV* array has been obtained, the parsing step can be performed within the time and space requirements due to Lemma 1. What remains is how to compute *NSV* using only a single integer array, including the *NSV* array itself.

Our algorithm achieves this in two steps. We first show in Section 4.1 that, given the Φ array, *NSV* can be computed in linear time and $O(1)$ extra space, by rewriting Φ array in-place. Then, we show in Section 4.2 that, given T, the Φ array can be computed in linear time and $O(1)$ extra space. By combining the two algorithms, we obtain our main result.

Theorem 1. *Assuming a constant size integer alphabet, the LZ77 factorization of a string of length N can be computed in $O(N)$ time using of $N \log N + O(1)$ bits of total working space, i.e., a single auxiliary integer array of length N.*

We call the algorithm that uses two integer arrays by incorporating the former technique, BGtwo, and the algorithm that uses only a single integer array by incorporating both techniques, BGone. (See Figure 1)

4.1 In-place computation of the *NSV* array from the Φ array

Since $\Phi[i]$ for each i indicates lexicographic predecessor of $suf(i)$, we can sequentially access values of *SA* from right to left, by accessing the Φ starting from the lexicographically largest suffix, which is $\Phi[0]$. More precisely, since the *SA* is a permutation of the integers $1, \ldots, N$, Φ can be regarded as an array based implementation of a singly linked list, linking the elements of *SA* from right to left. Thus, the algorithm for computing *NSV* from *SA* can be simulated using the Φ array. An important difference is that while elements of *SA* are in lexicographic order, elements of Φ are in text order, which is the same as *NSV*. Also, since the access on *SA* is sequential, the value $\Phi[i]$ is not required anymore after it is processed, and we can rewrite $\Phi[i]$ to $NSV[i]$. The pseudo code of the algorithm is shown in Algorithm 2. The correctness and running time follows from the above arguments.

Algorithm 2: In-place computation of *NSV* from Φ.

```
Input : Φ array (denoted as NSV)
1 cur ← NSV[0] ; // Φ[0]: lexicographically largest suffix
2 prev ← 0 ;
3 while cur ≠ 0 do
4     while cur < prev do
5         prev ← NSV[prev] ; // peak elimination
6     next ← NSV[cur] ; // Φ[cur]
7     NSV[cur] ← prev ;
8     prev ← cur ;
9     cur ← next ;
```

Lemma 2. *Given the Φ array of a string T, NSV array of T can be computed from Φ in linear time and in-place using $O(1)$ working space.*

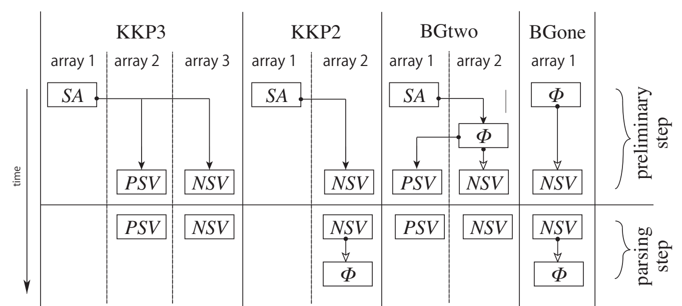

Fig. 1 A comparison of the auxiliary arrays used and how their contents change with time for the KKP variants and our algorithm.

4.2 Computing the Φ array using $O(1)$ working space

In the previous section, we showed that the *NSV* array can be computed from the Φ array in-place in linear time. By combining Lemma 1 and Lemma 2, if the Φ array is given, we can compute the LZ-factorization in linear time by rewriting Φ array to *NSV* array in-place, and rewriting *NSV* array to Φ array in-place (and sequentially obtain *NSV* and *PSV* values), using only constant extra working space. The problem is now how to compute the Φ array. Although the Φ array can easily be computed in linear time by a naive sequential scan on *SA*, storage for both the input *SA* and output Φ array is required for such an approach, as in the case of computing *NSV* from *SA*. As far as we know, an in-place linear time construction algorithm for the Φ array has not yet been proposed. Below, we propose the first such algorithm.

As noted in the previous subsection, the Φ array can be considered as an alternative representation of *SA*, which allows us to simulate a sequential scan on the *SA*. Thus, in order to construct Φ in-place, our algorithm simulates the in-place suffix array construction algorithm by Nong [13] which runs in linear time on

© 2013 Information Processing Society of Japan

constant size integer alphabets. We first describe the outline of the algorithm by Nong for computing *SA*, and then describe how to modify this to compute the Φ array.

4.2.1 Construction of the suffix array by induced sorting [13]

Nong's algorithm is based on induced sorting, which is a well known technique for linear time suffix sorting. Induced sorting algorithms first sort a certain subset of suffixes, either directly or recursively, and then induces the lexicographic order of the remaining suffixes by using the lexicographic order of the subset. There exist several methods depending on which subset of suffixes to choose. Nong's algorithm utilizes the concept of LMS suffixes defined below.

Definition 2. *For $1 \leq i \leq N$, a suffix $suf(i)$ is an L-suffix if $suf(i)$ is lexicographically larger than $suf(i+1)$, and an S-suffix otherwise. We call S or L the* type *of the suffix. An S-suffix $suf(i)$ is a Left-Most-S-suffix (LMS-suffix) if $suf(i)$ is an S-suffix and $suf(i-1)$ is an L-suffix.*

Recall that $T[N+1] = \$$, where \$ is a special delimiter character that does not occur elsewhere in the string. We define $suf(N+1)$ to be an S-suffix. Notice that for $i \leq N$, $suf(i)$ is an S-suffix iff $T[i] < T[i+1]$, or $T[i] = T[i+1]$ and $suf(i+1)$ is an S-suffix. The type of each suffix can be determined by scanning T from right to left.

In *SA*, all suffixes starting with the same character c occur consecutively, and we call the interval on the suffix array of such suffixes, the c-interval. A simple observation is that the L-suffixes that start with some character c must be lexicographically smaller than all S-suffixes that start with the same character c. Thus a c-interval can be partitioned into to two sub-intervals, which we call the L-interval and S-interval of c.

The induced sorting algorithm consists of the following steps. We denote the working array to be *SA*, which will become the suffix array of the text at the end of the algorithm.

(1) Sort the LMS-suffixes.
We call the result *LMS_SA*. We omit details of how this is computed, since our algorithm will use the algorithm described in [13] as is, but it may be performed in linear time using $O(1)$ extra working space. We assume that the result *LMS_SA* is stored in the first k elements of *SA*, i.e. $SA[1..k]$, where k is the number of LMS-suffixes.

(2) Put each LMS-suffix into the S-interval of its first character, in the same order as *LMS_SA*.
We scan T from right to left, and for each $c \in \Sigma$, compute and store the number of L-suffixes and S-suffixes, that start with c. We also compute the number of suffixes that start with a character that is lexicographically smaller than c. Storing these values requires only constant space, since we assume a constant size alphabet. From these values, we can determine the start and end positions of the L-interval and S-interval for any c. Initially, all intervals are marked empty. By also maintaining a pointer to the left-most or right-most empty element in an interval, adding elements to an L-interval or S-interval can also be performed in $O(1)$ time using $O(1)$ extra space. By a right to left scan on *LMS_SA* (i.e. $SA[1..k]$), we put each LMS-suffix in the right most empty element of the S-interval of the corresponding character

(3) Sort and put the L-suffixes in their proper positions in *SA*.
This is done by scanning *SA* from left to right. For each position i, if $SA[i] > 1$ and $suf(SA[i]-1)$ is an L-suffix, $suf(SA[i]-1)$ is put in the left-most empty position of the L-interval for character $T[SA[i]-1]$. The correctness of the algorithm follows from the fact that if suffix $suf(SA[i]-1)$ is an L-suffix, then, $suf(SA[i])$ must have been located before i (in the correct order), in *SA*.

(4) Sort and put the S-suffixes in their proper positions in *SA*.
This is done by scanning *SA* from right to left. For a position i, if $SA[i] > 1$ and $suf(SA[i]-1)$ is an S-suffix, $suf(SA[i]-1)$ is put in the right most empty position of the S-interval for character $T[SA[i]-1]$. The correctness of the algorithm follows from the fact that if suffix $suf(SA[i]-1)$ is an S-suffix, then, $suf(SA[i])$ must have been located after i (in the correct position), in *SA*.

In total, the algorithm computes suffix array in linear time using only a single integer array and constant extra working space. Note that for any position i, determining whether suffix $suf(SA[i]-1)$ is an L-suffix or not, can be done in $O(1)$ time using no extra space. If $T[SA[i]-1] < T[SA[i]]$ then it is an S-suffix, and if $T[SA[i]-1] > T[SA[i]]$ then it is an L-suffix. For the case of $T[SA[i]-1] = T[SA[i]]$, the type of $suf(SA[i]-1)$ is the same as that of $suf(SA[i])$, which can be determined by the position i, and the start and end positions of the L and S-intervals of character $T[SA[i]]$.

4.2.2 Construction of the Φ array by induced sorting

We regard Φ as an array based implementation of a singly linked list containing elements of *SA* from right to left. The basic idea of our algorithm to construct the Φ array is to modify Nong's algorithm for computing *SA*, to use this list representation instead. However, there are some technicalities that need to be addressed.

We denote the working array to be A, which will be an array based representation of a singly linked list that links (in lexicographic order) the set of so-far sorted suffixes at each step, and will become the Φ array of the text at the end of the algorithm. The algorithm is described below.

(1) Sort the LMS-suffixes.
First, we sort LMS-suffixes in the same way as [13]. The result will be called *LMS_SA* and stored in $A[1..k]$, where k is the number of LMS-suffixes.

(2) Put each LMS-suffix into the S-interval of its first character, in the same order as *LMS_SA*.
In this step, we transform *LMS_SA* to the array based linked list representation, so that for each LMS-suffix $suf(LMS_SA[i])$, its lexicographically succeeding LMS-suffix $suf(LMS_SA[i+1])$ will be put in $A[LMS_SA[i]]$, i.e., $A[LMS_SA[i]] = LMS_SA[i+1]$ for $i < k$. If *LMS_SA* and A were different arrays, then we could simply set $A[LMS_SA[i]] = LMS_SA[i+1]$ for each $i < k$. The problem here is that since *LMS_SA* is stored in $A[1..k]$, when setting a value at some position of A, we may overwrite a value of *LMS_SA* which has not been used yet. We overcome this problem as follows.

First, we memorize $LMS_SA[1]$, the first value of LMS_SA. Then, for $1 \le i \le k$, we set $A[2i] = LMS_SA[i]$ and $A[2i-1] = EMPTY$ by scanning $A[1..k]$ from right to left. Since k never exceeds $N/2$, we have $2i \le N$ for all $1 \le i \le k$. Next, for $1 \le i \le k-1$, let $j_1 = A[2i](= LMS_SA[i])$ and $j_2 = A[2(i+1)](= LMS_SA[i+1])$. We attempt to set $A[j_1] = j_2$. If $A[j_1] = EMPTY$, then we simply set $A[j_1] = j_2$. Otherwise $j_1 = 2i'$ for some $1 \le i' \le k$, and $A[j_1]$ stores the value $LMS_SA[i']$. Therefore, we do not overwrite this value, but instead, borrow the space immediately preceding position j_1, and set $A[j_1 - 1] = j_2$. An important observation is that $A[j_1 - 1]$ must have been $EMPTY$, because LMS-suffixes cannot, by definition, start at consecutive positions, and if j_1 was an LMS suffix, $j_1 - 1$ cannot be an LMS suffix and the algorithm will never try to set another value at this position.

After this, we set $A[2i] = EMPTY$ for all $1 \le i \le k$, and we arrange the remaining values to their correct positions by attempting to traverse succeeding suffixes stored in A from the lexicographically smallest suffix of LMS_SA memorized at the beginning of the process. Let i be the current position we are traversing. We attempt to obtain its succeeding suffix by reading $A[i]$. If $A[i] \ne EMPTY$, the succeeding suffix of $suf(i)$ was stored at correct position, and we continue with the next position $A[i]$. If $A[i] = EMPTY$, then the succeeding suffix of $suf(i)$ may be stored at the immediately preceding position, i.e. $A[i-1]$. In such a case, $A[i-1] \ne EMPTY$, and we set $A[i] = A[i-1]$ and $A[i-1] = EMPTY$, and continue with the next position $A[i]$. If $A[i-1] = EMPTY$, this means that $suf(i)$ is the lexicographically largest suffix of LMS-suffixes, and we finish the process.

In this way, for all LMS-suffixes $suf(i)$, we can set the succeeding suffix at $A[i]$. The process essentially scans the values of LMS_SA on A twice. Therefore, this step runs in $O(k)$ time and $O(1)$ working space.

(3) Sort and put the L-suffixes in their proper positions in A.

To simulate the algorithm for SA, we need to scan the suffixes in lexicographically increasing order by using A. Let $suf(i)$ be a suffix the algorithm is processing. We want to set $A[j] = i-1$ if $suf(i-1)$ is an L-suffix, and $suf(j)$ is the suffix that lexicographically precedes suffix $suf(i-1)$.

To accomplish this, we introduce four integer arrays of size $|\Sigma|$ each, $Lbkts[c]$, $Lbkte[c]$, $Sbkts[c]$ and $Sbkte[c]$. $Lbkts[c]$ and $Lbkte[c]$ store the lexicographically smallest and largest suffix of the L-interval for a character c which have been inserted into A, and $Sbkts[c]$ and $Sbkte[c]$ are the same for each S-interval. All values are initially set to $EMPTY$. We first scan the list of LMS suffixes in lexicographically increasing order represented in A constructed in the previous step, and insert each LMS suffixes into the corresponding S-interval, by updating $Sbkts[c]$ and $Sbkts[e]$. Then, we scan all LMS- and L-suffixes in lexicographically increasing order by traversing the succeeding suffixes on A by starting from $Lbkts[c]$, traversing the list represented by A until we process $Lbkte[c]$. Then we do the same starting from $Sbkts[c]$ and process the suffixes until we reach $Sbkte[c]$, and repeat the process for all character c in lexicographic order.

Let $suf(i)$ be a suffix the algorithm is currently processing. We store $suf(i-1)$ in the appropriate position of A, if $suf(i-1)$ is an L-suffix, and do nothing otherwise. Since we know the type of suffix $suf(i)$ since we are either processing a suffix between $Lbkts[c]$ and $Lbkte[c]$ or $Sbkts[c]$ and $Sbkte[c]$, the type of $suf(i-1)$ can be determined in constant time by simply comparing $T[i-1]$ and $T[i]$, i.e. it is an L-suffix if $T[i-1] > T[i]$, an S-suffix if $T[i-1] < T[i]$, and has the same type as $suf(i)$ if $T[i-1] = T[i]$.

When storing $suf(i-1)$ in A, we check $Lbkts[T[i]]$. If $Lbkts[T[i-1]] = EMPTY$, then, $suf(i-1)$ is the lexicographically smallest suffix starting with $T[i-1]$. We set $Lbkts[T[i-1]] = Lbkte[T[i-1]] = i-1$. Otherwise, there is at least one suffix lexicographically smaller than $suf(i-1)$ in the L-interval for character $T[i-1]$. This suffix is $Lbkte[T[i-1]] = j$, and we set $A[j] = i-1$, and update $Lbkte[T[i-1]] = i-1$.

In this way we can compute all the lexicographically succeeding suffix of each L-suffixes in the corresponding L-interval, and store them in A. Since the number of times we read the values of A is at most the number of LMS- and L-suffixes, and the updates for each new L-suffix can be done in $O(1)$ time, the algorithm runs in linear time using only a single integer array and $O(1)$ working space in total.

(4) Sort and put the S-suffixes in their proper positions in A.

To simulate the algorithm for SA, we need to scan all L-suffixes in lexicographically decreasing order by using A. However, since the linked list of L-suffixes constructed on A in the previous step is in increasing order, we first rewrite A to reverse the direction of the links. That is, we want to set $A[j] = i-1$ if $suf(i-1)$ is an L-suffix and $suf(j)$ is the suffix that lexicographically succeeds suffix $suf(i-1)$.

This rewriting can be done by scanning the succeeding suffixes in a similar way as that of Step 3. For each c in lexicographically increasing order, traverse the L-suffixes by using $Lbkts[c]$, $Lbkte[c]$, and A, and simply rewrite the values in A to reverse the links, i.e., if $suf(j)$ preceded $suf(i)$ then $A[i] = j$.

Now we have a lexicographically decreasing list of L-suffixes represented in A, and want to insert the S-suffixes into A. The process is similar to that of Step 3. Initially the values for $Sbkts[c]$ and $Sbkte[c]$ for all c are set to $EMPTY$. Then, for each c in lexicographically *decreasing* order, we traverse preceding suffixes on A by starting from $Sbkte[c]$, traversing the list represented by A until we process $Sbkts[c]$. Then we do the same starting from $Lbkte[c]$ and process the suffixes until we reach $Lbkts[c]$, and so on. Let $suf(i)$ be a suffix the algorithm is currently processing. If $suf(i-1)$ is an S-suffix, we store $suf(i-1)$ in the appropriate position of A and update $Sbkts[c]$ and $Sbkte[c]$ accordingly, and do nothing otherwise. A minor detail during this process is that we also link preceding suffixes which are in different S or L intervals.

Now that all suffixes have been inserted and linked, we can obtain all suffixes in decreasing order by traversing preced-

© 2013 Information Processing Society of Japan

ing suffixes on A, i.e. A is now equal to the Φ array. Similarly to the previous step, we can see that this step runs in linear time using one integer array of length N (A) and $O(1)$ extra space.

All steps run in linear time using A and $O(1)$ extra space, thus giving a linear time algorithm for computing Φ using $O(1)$ extra working space.

The above procedure describes how to construct Φ from T using only a single integer array of length N. We propose another variant of the algorithm that, given SA, computes the Φ by rewriting SA in-place in linear time and $O(1)$ extra working space. The idea may seem useless at a glance, but may have applications when the SA is already available, since the conversion does not require the expensive recursion step as in the linear time SA construction algorithm (in Step 1), but can be achieved in a few scans.

Lemma 3. *Given the SA of a string T of length N, Φ array of T can be computed from SA in $O(N)$ time and in-place using $O(1)$ working space.*

Proof. It suffices to compute LMS_SA, since then we can run the above algorithm from Step 2. We scan T from right to left, and for each character c, count the number of L- and S-suffixes that start with c, and obtain the L- and S-interval for each character c on SA. Let k be a counter of the number of LMS suffixes initially set to 0. We then scan SA from left to right for $1 \leq i \leq N$. If i is within an S-interval and $T[SA[i]] < T[SA[i]-1]$, then, $suf(SA[i])$ is an LMS-suffix and we store it in $SA[k+1]$, and increment k.

In this way, we can obtain LMS_SA and also SA by applying Step 2-4 in $O(N)$ time and $O(1)$ extra working space. □

4.3 In-place computation of SA from the Φ array

An advantage of the KKP algorithms compared to BGone may be that SA is left untouched after the LZ-factorization. On the other hand, the Φ array is left after running BGone. Actually, it is possible to show that the Φ array can be converted back to SA in linear time and in-place, using $O(1)$ extra working space.

Lemma 4. *Given a string T and its Φ array, the SA array of T can be computed in linear time and in-place using $O(1)$ working space.*

Proof. The induced sorting algorithm constructs SA by first computing LMS_SA and stores it in $SA[1..k]$, where k is the number of LMS suffixes. Thus, if we can somehow compute LMS_SA from the Φ array in linear time using $O(1)$ extra working space and save it in $SA[1..k]$, we have proved the lemma.

Let A be an integer array of size N, used in our algorithm, initially equal to the Φ array. Our algorithm will consist of two steps. First, for all LMS-suffixes $suf(i)$, we compute the preceding suffix of $suf(i)$, and store it in $A[i]$ (we store $A[i] = EMPTY$ if $suf(i)$ is not an LMS suffix), thus obtaining an array based linked list representation of LMS-suffixes in lexicographically decreasing order. Second, we rewrite A so that $A[1..k] = LMS_SA[1..k]$, reversing the procedure described in Step 2 of Section 4.2.2.

For the first step, we compute for each character c, the starting positions in SA of the S-interval for c by counting the number of L-suffixes and S-suffixes that start with the character c. As in Step 2 of Section 4.2.1, this can be done in linear time and constant space. We then simulate a right to left traversal on the SA using the Φ array stored as A. Let $suf(SA[i])$ be the suffix that the algorithm currently processing. For $suf(SA[i])$ to be an LMS-suffix, it must be that $suf(SA[i])$ is an S-suffix, and also $T[SA[i]-1] > T[SA[i]]$. The former condition can be checked by whether the position i is in an L-interval or an S-interval. During the process, we remember the previous LMS-suffix $suf(j)$, and set $A[j] = i$ if $suf(i)$ is an LMS-suffix, and we continue traversing by reading $A[i]$ and setting $A[i] = EMPTY$. In this way, we can compute a lexicographically decreasing list of LMS suffixes, represented in A in linear time and $O(1)$ working space.

Now, we only have to rearrange this list of suffixes to LMS_SA. The process is the opposite of Step 2 in Section 4.2.2. We first traverse the LMS suffixes in lexicographically decreasing order. We try to set the largest LMS suffix at $A[2k]$, the second largest LMS suffix at $A[2(k-1)]$ and so on. If for the ith largest LMS suffix, $A[2i] = EMPTY$, we simply set $A[2i]$ to be this value. Otherwise, $2i$ was an LMS-suffix and part of the list. In this case, we store the value in $A[2i-1]$. Notice that again since LMS suffixes cannot start at consecutive positions, if $2i$ was an LMS suffix, $2i-1$ cannot be an LMS suffix, and the algorithm will never try to set another value at this position.

Since the original linked list of LMS-suffixes was not overwritten and is preserved, we can traverse this again this time setting the corresponding positions to $EMPTY$. Then, checking all positions $2i$ for $1 \leq i \leq k$, if $A[2i] = EMPTY$ then the corresponding value was stored in $A[2i-1]$ and can be retrieved. Finally, we copy the values at $A[2i]$ to $A[i]$ for each $1 \leq i \leq k$. Thus, LMS_SA can be computed in linear time using $O(1)$ working space. □

5. Computational Experiments

We implemented BGtwo and two variations of BGone, these are differ in the computation of Φ array. One of which computes Φ array directly from T (BGoneT), and the other firstly computes SA and then computes Φ array from SA (BGoneSA). The 3 implementation are available at `http://code.google.com/p/bgone/`. We compared our algorithms with the implementation of KKP1, KKP2, and KKP3 [*2]. We use SACA-K which is the implementation of Nong's algorithm to compute LMS_SA in BGoneT, and use divsufsort to compute SA in the other implementations, BGtwo, BGoneSA, KKP1, KKP2, and KKP3. Note that in terms of speed, BGoneT has a disadvantage since although divsufsort is not a truly linear time algorithm, it is generally faster than SACA-K. These conditions were chosen since the latter algorithms can choose any suffix array construction algorithm, while BGoneT cannot.

All computations were conducted on a Mac Xserve (Early 2009) with 2 x 2.93GHz Quad Core Xeon processors and 24GB Memory, only utilizing a single process/thread at once. The programs were compiled using the GNU C++ compiler (`g++`) 4.7.1 with the `-Ofast -msse4.2` option for optimization. The running times are measured in seconds, starting from after reading input text in memory, and the average of 3 runs is reported. We

[*2] `https://www.cs.helsinki.fi/group/pads/lz77.html`

© 2013 Information Processing Society of Japan

Table 1 Time and space consumption for computing LZ factorization. The times were measured after reading input text in memory. The runtime of KKP1 includes the writing and reading time of *SA* to and from the disk.

Algo	KKP1	KKP2	KKP3	BGtwo	BGoneT	BGoneSA
# Arrays	2	2	3	2	1	1
pro	75.57	67.50	56.41	80.30	157.85	136.13
eng	69.34	61.11	49.84	75.50	156.25	132.29
dna	72.35	64.01	52.76	79.46	146.46	136.85
src	55.58	47.45	38.53	57.68	116.18	98.76
cor	54.83	46.68	37.45	58.64	116.94	101.34
cere	140.82	122.83	107.51	177.94	323.00	299.18
ker	69.19	59.04	49.81	77.15	154.16	131.48
ein	145.25	126.92	111.90	178.67	333.97	287.76

use the data used in previous work [*3]. Table 5 shows running times of the algorithms, and how many integer arrays is used.

The results show that the runtimes of our algorithms is only about twice as slow as KKP1, despite the added complexity introduced so that the algorithm can run on a single integer array. One reason that KKP1 is faster may be because BGone needs random access on the integer array to compute the *NSV* array, while KKP1 does not. Although KKP1 needs to write and read *SA* to and from the disk, sequential I/O seems to be faster than random access on the memory. BGoneSA which computes Φ array through *SA* is a little faster than BGoneT which computes Φ directly.

References

[1] Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylov, J., Smyth, W., Tischler, G. and Yusufu, M.: A comparison of index-based Lempel-Ziv LZ77 factorization algorithms, *ACM Computing Surveys* (in press).

[2] Crochemore, M.: Linear searching for a square in a word, *Bulletin of the European Association of Theoretical Computer Science*, Vol. 24, pp. 66–72 (1984).

[3] Crochemore, M. and Ilie, L.: Computing Longest Previous Factor in linear time and applications, *Information Processing Letters*, Vol. 106, No. 2, pp. 75–80 (2008).

[4] Duval, J.-P., Kolpakov, R., Kucherov, G., Lecroq, T. and Lefebvre, A.: Linear-time computation of local periods, *Theoretical Computer Science*, Vol. 326, No. 1-3, pp. 229–240 (2004).

[5] Goto, K. and Bannai, H.: Simpler and Faster Lempel Ziv Factorization, *DCC*, pp. 133–142 (2013).

[6] Kärkkäinen, J., Kempa, D. and Puglisi, S. J.: Linear Time Lempel-Ziv Factorization: Simple, Fast, Small, *Proc. CPM'13* (2013).

[7] Kärkkäinen, J., Manzini, G. and Puglisi, S. J.: Permuted Longest-Common-Prefix Array, *CPM*, pp. 181–192 (2009).

[8] Kärkkäinen, J. and Sanders, P.: Simple Linear Work Suffix Array Construction, *Proc. ICALP 2003*, pp. 943–955 (2003).

[9] Kasai, T., Lee, G., Arimura, H., Arikawa, S. and Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, *Proc. CPM 2001*, pp. 181–192 (2001).

[10] Kolpakov, R. and Kucherov, G.: Finding Maximal Repetitions in a Word in Linear Time, *Proc. FOCS 1999*, pp. 596–604 (1999).

[11] Kreft, S. and Navarro, G.: Self-indexing Based on LZ77, *Proc. CPM 2011*, LNCS, Vol. 6661, pp. 41–54 (2011).

[12] Manber, U. and Myers, G.: Suffix arrays: A new method for on-line string searches, *SIAM J. Computing*, Vol. 22, No. 5, pp. 935–948 (1993).

[13] Nong, G.: Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets, *ACM Trans. Inf. Syst.*, Vol. 31, No. 3, p. 15 (2013).

[14] Nong, G., Zhang, S. and Chan, W. H.: Two Efficient Algorithms for Linear Time Suffix Array Construction, *IEEE Trans. Computers*, Vol. 60, No. 10, pp. 1471–1484 (2011).

[15] Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression, *Theoretical Computer Science*, Vol. 302, No. 1–3, pp. 211–222 (2003).

[16] Ziv, J. and Lempel, A.: A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory*, Vol. IT-23, No. 3, pp. 337–343 (1977).

[*3] `http://pizzachili.dcc.uchile.cl/texts.html`, `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

© 2013 Information Processing Society of Japan