Computing the Inverse Sort Transform in Linear Time

GE NONG, Sun Yat-sen University
SEN ZHANG, SUNY College at Oneonta
WAI HONG CHAN, Hong Kong Baptist University

The Sort Transform (ST) can significantly speed up the block sorting phase of the Burrows-Wheeler Transform (BWT) by sorting the limited order contexts. However, the best result obtained so far for the inverse ST has a time complexity $O(N \log k)$ and a space complexity $O(N)$, where N and k are the text size and the context order of the transform, respectively. In this article, we present a novel algorithm that can compute the inverse ST for any k-order contexts in an $O(N)$ time and space complexity, a linear result independent of k. The main idea behind the design of this linear algorithm is a set of cycle properties of k-order contexts that we explore for this work. These newly discovered cycle properties allow us to quickly compute the Longest Common Prefix (LCP) between any pair of adjacent k-order contexts that may belong to two different cycles, which eventually leads to the proposed linear-time solution.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Burrows-Wheeler transform, inverse transform, limited order context, linear time

ACM Reference Format:
Nong, G., Zhang, S., and Chan, W. H. 2011. Computing the inverse sort transform in linear time. ACM Trans. Algor. 7, 2, Article 27 (March 2011), 16 pages.
DOI = 10.1145/1921659.1921673 http://doi.acm.org/10.1145/1921659.1921673

1. INTRODUCTION

The Burrows-Wheeler Transform (BWT), introduced in 1994 [Burrows and Wheeler 1994], has been successfully used in a wide range of data compression applications as an efficient preprocessing component [Adjeroh et al. 2002, 2008; Arnavut 1999, 2001; Mukherjee et al. 2001; Schindler and Sebastian 2001]. Briefly, given a text S of N characters, the BWT can be principally performed in three steps: (1) derive a matrix consisting of N rotations (cyclic shifts) of S, which is called the rotation matrix; (2) sort the rows of the matrix lexicographically; and (3) extract the last column of the sorted matrix to produce the transformed text. Although the BWT itself does not reduce the

---

The research of G. Nong was supported by the National Natural Science Foundation of China (grant no. 60873056) and the Fundamental Research Funds for the Central Universities of China (grant no. 111gzd04). The work of W. H. Chan was partially supported by CERG (HKBU210207), Research Council, Hong Kong.
Authors' addresses: G. Nong, Computer Science Department, Sun Yat-sen University, Guangzhou 510275, P.R.C.; email: issng@mail.sysu.edu.cn; S. Zhang, Department of Mathematics, Computer Science and Statistics, SUNY College at Oneonta, Oneonta, NY 13820; email: zhangs@oneonta.edu; W. H. Chan, Department of Mathematics, Hong Kong Baptist University, Hong Kong; email: dchan@hkbu.edu.hk.

DOI 10.1145/1921659.1921673 http://doi.acm.org/10.1145/1921659.1921673

length of a given text, the transformed text tends to group identical characters together so that the probability of finding them clustered together is increased substantially. Inspired by the success of the BWT, many variants have also been proposed by the research community during the past decade. Among them, one noticeable is the Sort Transform (ST) that was introduced in 1997 by Schindler [1997, 2000], which can speed up the block sorting phase of the BWT by sorting only a portion of the rotation matrix. The ST is built upon the concept of limited and unlimited order contexts. Let the preceding characters be the context of the last character of a text. A *k-order context*, where $k \in [1, N]$, is a context containing only the first k characters of the text. In the case of $k = N$, the full text (including the last character) is the context, hence its order is deemed as unlimited. Otherwise, the order of the context is regarded as limited.

The main idea of the ST is to limit sorting to the first k columns only, instead of the full matrix sorted by the BWT. Specifically, given the same rotation matrix as defined in the BWT, the k-order ST will lexicographically sort all the rows of the matrix according to their k-order contexts first; in case that there are any two identical k-order contexts, the tie will be resolved by preserving the relative order between them in the original rotation matrix. In other words, the sorting is stable, that is, for any two identical contexts, their order in the sorted matrix must be kept as that in the original matrix. As a consequence, the ST resembles the BWT but the reduced context comparison length (actually the reduced sorting problem size) accelerates the sorting phase, which makes it a noticeably faster alternative to the BWT [Adjeroh et al. 2008; Manzini 1999; Balkenhol et al. 1999; Baron and Bresler 2005; Yokoo 1999]. Puglisi et al. [2005] showed by experiments that the superior asymptotic complexity of recently developed linear-time suffix array algorithms [Karkkainen and Sanders 2003; Ko and Aluru 2003; Kim et al. 2003] does not readily translate into faster suffix sorting, when compared to the superlinear algorithms [Manzini and Ferragina 2004; Larsson and K.Sadakane 1999] including those based on simple radix-sort variants. With only a relatively small adjustment to the sorting size of the matrix, the ST can be expected to perform much faster than the BWT, yet still retaining a high compression ratio. Schindler has built a fast compression software called `szip` [Schindler 1999] using the ST.

1.1. Statement of the Problem

A major trade-off caused by the ST partial sorting scheme is that the inverse ST is more complex than the inverse BWT. This is because although each unlimited context is unique, the uniqueness of any limited order context considered by the partial sorting scheme in ST can no longer be guaranteed. To deal with the duplicated k-order contexts, Schindler proposed a hash table-based approach which has an $O(kN)$ time and space complexity. In this approach, the text retrieval has to rely on a hash table-driven context lookup and the context lookup has to rely on the complete restoration of all the k-order contexts [Schindler 2001]. Noticing that neither the full restoration of contexts nor the hash-based context lookup is required by the inverse BWT, we have proposed an auxiliary vectors-based framework [Nong and Zhang 2006, 2007a, 2007b] which is similar to that used for the inverse BWT [Burrows and Wheeler 1994], but different from any possible hash table-based approaches suggested by Schindler [2001], Yokoo [1999], and Bird and Mu [2004]. While this framework does efficiently reduce the space complexity to $O(N)$, the time complexities of our previous solutions remain to be superlinear. The time complexity achieved in Nong and Zhang [2006] is $O(kN)$, which we have recently reduced to $O(N \log k)$ in Nong and Zhang [2007a, 2007b]. All the time complexities of the existing solutions for the inverse ST involve the context order k, one way or another. In contrast, the inverse BWT has a linear time and space complexity of $O(N)$. Therefore, our particular interest here is whether the inverse ST can be computed in linear time.

M_2	R	P_2	Q_2	D_2	C_2	T_2
i p p i $ m i s s i s s	1	8	7	1	1	8
i s s i s s i p p i $ m	2	5	10	1	2	5
i s s i p p i $ m i s s	3	9	11	0	0	8
i $ m i s s i s s i p p	4	6	12	1	1	6
m i s s i s s i p p i $	5	12	2	1	1	12
p i $ m i s s i s s i p	6	7	4	1	1	7
p p i $ m i s s i s s i	7	1	6	1	1	1
s i s s i p p i $ m i s	8	10	1	1	2	10
s i p p i $ m i s s i s	9	11	3	0	0	10
s s i s s i p p i $ m i	10	2	8	1	2	2
s s i p p i $ m i s s i	11	3	9	0	0	2
$ m i s s i s s i p p i	12	4	5	1	1	4

Fig. 1. M_2 is the 2-order sorted matrix, where all the rows have been sorted according to their 2-order contexts. The last column of M_2 is the transformed text, and the starting index is 5.

1.2. Our Solution

In this article, we answer this question by presenting a novel algorithm that explores a set of properties about Longest Common Prefixes (LCPs) and cycles in the k-order contexts, to compute the inverse ST for any context order $k \in [1, N]$. The new algorithm has a linear time and space complexity of $O(N)$, which is independent of k. This algorithm closes the theoretical gap standing for long between the inverse BWT and the inverse ST, and brings us new opportunities to build high-performance compression applications based on the ST.

The rest of this article is organized as follows. Section 2 introduces some basic definitions and general notations. Our linear-time inverse ST algorithm is developed and analyzed in Section 3 and 4, respectively. Section 5 gives the conclusion.

2. PRELIMINARY

Suppose S is a text of length N, and denoted as $S[1]S[2]S[3]\dots S[N]$, where each character $S[i] \in \Sigma$, $i \in [1, N]$ and Σ is a constant alphabet. The last character $S[N]$ is \$, a *sentinel*, which is the lexicographically greatest[1] and unique character in S. Furthermore, we call $S[i]$ the immediate preceding character of $S[i+1]$ where $i \in [1, N-1]$, and $S[N]$ the immediate preceding character of $S[1]$. By cyclic rotating S a total number of N times, we obtain the rotation matrix M_0 for computing the ST of S.

Definition 2.1 (*The rotation matrix M_0*). The $N \times N$ symmetric matrix consisting of N texts S_1 to S_N is obtained by rotating S. Specifically, the first row of the matrix M_0 is S and denoted by S_1; and for each of the remaining rows, a new text S_i is obtained by cyclically shifting the previous text S_{i-1} one column to the left. That is, $S_i = S_{i-1}[2]S_{i-1}[3]\cdots S_{i-1}[N]S_{i-1}[1]$.

Each row in M_0 is regarded as a text in which the first k characters is called the k-order context of the last character, for $k \in [1, N]$. As mentioned in Section 1, when $k = N$, the k-order context is also called the unlimited order context; otherwise, it is called a limited order context. Lexicographically *stable* sorting all the rows of M_0 by their k-order contexts, we will get a k-order sorted matrix M_k, whose last column is the k-order ST of S. For example, we give in Figures 1 and 2 the 2-order ST and BWT on a sample text $S = [\texttt{m i s s i s s i p p i \$}]$, respectively. In these two figures, M_2

[1]Symmetrically, the sentinel can be assumed as the smallest. Appending a sentinel to the original text has been used in many previous publications. Readers who are interested in the details of using a sentinel may refer to papers Manzini [1999], Balkenhol and Kurtz [2000], and Deorowicz [2002], since how to use a sentinel is not the main concern of this article.

M												R	P	Q
i	p	p	i	$	m	i	s	s	i	s	s	1	8	7
i	s	s	i	p	p	i	$	m	i	s	s	2	9	10
i	s	s	i	s	s	i	p	p	i	$	m	3	5	11
i	$	m	i	s	s	i	s	s	i	p	p	4	6	12
m	i	s	s	i	s	s	i	p	p	i	$	5	12	3
p	i	$	m	i	s	s	i	s	s	i	p	6	7	4
p	p	i	$	m	i	s	s	i	s	s	i	7	1	6
s	i	p	p	i	$	m	i	s	s	i	s	8	10	1
s	i	s	s	i	p	p	i	$	m	i	s	9	11	2
s	s	i	p	p	i	$	m	i	s	s	i	10	2	8
s	s	i	s	s	i	p	p	i	$	m	i	11	3	9
$	m	i	s	s	i	s	s	i	p	p	i	12	4	5

Fig. 2. An example M for the BWT, where the last column is the transformed text.

and M are the sorted matrices, where the index of each row is indicated by the column R, and the other remaining columns will be explained later. By taking the transpose of the last column of M_2 and locating the row position of the original text S, which is 5 in this case, the transformed result can be denoted as a couplet of the transformed text and the starting index,[2] that is, $ST(S, 2) = ([\texttt{s m s p \$ p i s s i i i}], 5)$. If we increase k to 12, for this sample text S, $ST(S, k)$ will agree with the BWT in Figure 2.

Let Z be a two-dimensional array consisting of N_r rows and N_c columns. For presentation simplicity, we use the notation $Z[a : b, c : d]$ to represent a two-dimensional subarray of Z which covers the rows from a to b and the columns from c to d, where $1 \leq a \leq b \leq N_r$ and $1 \leq c \leq d \leq N_c$. In the case of $a = b$ or $c = d$, the simpler forms $Z[a, c : d]$ or $Z[a : b, c]$ can be used instead, respectively. Further, we use $Z[a, c]$ when $a = b$ and $c = d$. In M_k, we have $F_k = M_k[1 : N, 1]^T$ and $L_k = M_k[1 : N, N]^T$, that is, the transposes of the first and the last columns of M_k, respectively, where $k \in [1, N]$. When $k = N$, we will write M, F and L for M_k, F_k and L_k, respectively.

3. THE LINEAR INVERSE ST ALGORITHM

3.1. Basic Notation and Terminology

For convenience of exposition, we define two vectors to establish a one-to-one mapping among the characters of F_k and L_k, as follows.

Definition 3.1 (P_k *and* Q_k). P_k and Q_k are two size-N row vectors, where the former satisfies:

(1) $F_k[P_k[i]] = L_k[i]$, for $i \in [1, N]$; and
(2) $P_k[i] < P_k[j]$ if $i < j$, for $L_k[i] = L_k[j]$;

and the latter satisfies:

(1) $L_k[Q_k[i]] = F_k[i]$, for $i \in [1, N]$; and
(2) $Q_k[i] < Q_k[j]$ if $i < j$, for $F_k[i] = F_k[j]$.

P_k maps the index of each character in L_k to its index at F_k, and Q_k maps the index of each character in F_k to its index at L_k. Furthermore, P_k and Q_k are reciprocal to each other, that is, $Q_k[P_k[i]] = i$ and $P_k[Q_k[i]] = i$. When $k = N$, the unsubscripted forms of P and Q can be used for P_k and Q_k instead, where P is also the LF-mapping introduced

[2] When the sentinel $ is assumed as the unique greatest (or smallest) character in S, we can simply denote $ST(S, k)$ using only the transformed text without the starting index. However, without such a sentinel assumption, the starting index may be necessary. For this reason, we keep using the couplet denotation for ST, even through the starting index is redundant given a sentinel.

in Burrows and Wheeler [1994] and Q is also the Φ function introduced in Grossi and Vitter [2005], respectively.

The original inverse BWT algorithm in Burrows and Wheeler [1994] was developed based on a combined use of the following properties.

PROPERTY 3.2. *Given L, array F can be obtained by sorting all the characters of L.*

PROPERTY 3.3. *The relative orders of identical characters in F and L are the same.*

PROPERTY 3.4. *Given $S[j] = L[i]$, we have $S[j-1] = L[P[i]]$ for $j \in [2, N]$, and $S[N] = L[P[i]]$ for $j = 1$.*

From Property 3.2 and 3.3, it is not difficult to see that P can be computed by traversing L at most twice. Property 3.4 guarantees that we can always backward retrieve the immediate predecessor for a restored character $S[j]$. Recalling the definition of P, we know that for a given index i, $P[i]$ should be the index to be traced back to, by which we can locate the character $L[P[i]]$, that is, the immediate preceding character of the character $L[i]$ in the original text S. To locate the preceding character of $L[P[i]]$, we can further retrieve it as $L[P[P[i]]]$. By repeating this process $N-2$ more times to get $L[P[P[P[i]]]]$, $L[P[P[P[P[i]]]]]$ and so on, we will restore the whole string S.

For example, the column P in Figure 2 shows the one-to-one mapping from L to F. Starting from row 5, we can simply follow P to chain up the characters at L to restore the original text using the inverse BWT algorithm. However, due to the partial sorting strategy of ST, Property 3.4 no longer holds for the ST, that is, the P_k vector does not warrant a correct restoration any more. This is because that the P_k mapping now indicates a row of the matrix that has the correct context but not necessarily the correct position. In fact, if we simply follow P_k to restore the string, we will face an early termination problem. For example, using P_2 in Figure 1, the process restores `missippi$` with `ssi` being missed. This is because P_2 does not provide an exact mapping of the characters in S from L_2 to F_2, due to the confusion caused by the duplicated k-order contexts. This brings a tough challenge to the inverse ST problem: given $L_k[i]$, how to locate its true immediate predecessor among *multiple candidates that share exactly the same k-order context*?

3.2. Algorithm Framework

Figure 3 recapitulates the auxiliary vector-based framework proposed in Nong and Zhang [2006, 2007b] which inverts ST using no hash table. This algorithm framework consists of four modules. In the first module, we start with obtaining F_k by simply sorting the characters of L_k. Then P_k and Q_k can be computed at the same time by simply traversing L_k and F_k once. In the second module, we use L_k and Q_k to compute the *k-order context switch vector D_k*, where each item of D_k is either 0 or 1 as defined next.

Definition 3.5 (*k-Order Context Switch Vector D_k*). Let $CT_k[i]$ be the k-order context of $L_k[i]$ in M_k, that is, $CT_k[i] = M_k[i, 1:k]$. D_k is a size-N row vector with each $D_k[i]$, $i \in [1, N]$ defined as

$$D_k[i] = \begin{cases} 0, & \text{for } i \in [2, N] \text{ and } CT_k[i] = CT_k[i-1]; \\ 1, & \text{otherwise.} \end{cases}$$

In the third module, after constructing D_k to locate all the groups of identical k-order contexts, we obtain two size-N vectors, C_k and T_k, where C_k records the number of occurrence of each k-order context at its first position in the M_k while T_k indicates the starting row of each group of identical k-order contexts. We call C_k the *counter vector*. If $i = 1$ or $CT_k[i] \neq CT_k[i-1]$ for $i \in [2, N]$, we set $C_k[i]$ be the number of repetitions of

```
    GIBWT(L_k, I, k)
       ▷ Module 1
 1     Compute F_k, P_k and Q_k from L_k;

       ▷ Module 2
 2     Compute D_k;

       ▷ Module 3 calculates the frequency and the starting point of each context.
 3     Initialize all the items of counter array C_k[1, N] as 0;
 4     for i ← 1 to N
 5         do
 6            if D_k[i] = 1
 7               then j ← i;
 8            T_k[Q_k[i]] ← j;
 9            C_k[j] ← C_k[j] + 1;

       ▷ Module 4 restores the original text S from L_k.
10     j ← I;
11     for i ← N downto 1
12         do
13            S[i] ← L_k[j];
14            j ← T_k[j];
15            C_k[j] ← C_k[j] − 1;
16            j ← j + C_k[j];   ▷ Update the index for next character.

17     return S;
```

Fig. 3. The proposed GIBWT (Generalized Inverse BWT) algorithm framework for inverting ST.

$CT_k[i]$ starting from the ith row; otherwise, we set $C_k[i] = 0$. For example, in Figure 1, $C_2[2] = 2$ and $C_2[3] = 0$ because $CT_2[2]$ and $CT_2[3]$ are the same. T_k is called the *index vector*, where $T_k[i]$ is pointing to the smallest row having $L_k[i]M_k[i, 1 : k-1]$ as a context. Given $L_k[i]$, T_k tells that starting from the $T_k[i]$th row in M_k, there are $C_k[T_k[i]]$ consecutive rows sharing the same k-order context which is made up of $L_k[i]M_k[i, 1 : k-1]$. For example, in Figure 1, both $T_2[1]$ and $T_2[3]$ are set to 8, because row 8 starts with a group of two same k-order contexts.

In the fourth module, T_k and C_k are used together to restore the original text S from L_k in a way reminiscent of the classical BWT inverse algorithm [Burrows and Wheeler 1994]. Using the transformed result obtained in Figure 1 as an example, we now show how to dynamically recover the true index for the immediate predecessor of any given row through a combined use of T_2 and C_2.

Since L_2 is only a permutation of S, to recover S from L_2, we only need to keep generating a one-to-one mapping between the characters of the two strings in an orderly way. The algorithm works backwardly, that is, the retrieval process starts from the last position. At the beginning, the retrieval position index i is initialized to be 12. We know the last character of S must be "\$". The corresponding position of j from L_2 is initialized directly by I which is already known to be 5. I points to the last character "\$" which turns out to be the first to be retrieved without any ambiguity. To continue the mapping process, we need to update both i and j in such a way that they always point to the next characters in S and L_2, respectively. Update of i isn't an issue, because i simply needs to decrement by 1 step by step to mimic the backward retrieval process. But calculating next j from the current j needs a combined use of T_2 and C_2 (refer to lines 14–16 in Figure 3). We first update j using the formula $j = T_2[j] = T_2[5] = 12$. Because that duplicated contexts may exist in ST, $j = 12$ in this case only points to the start position of the (possible) cluster of multiple candidates that share the same 2-order context of $L_2[12]$. Since C_2 counts how many duplicated contexts have not been visited so far, we can update $C_2[12] = C_2[12] - 1 = 1 - 1 = 0$. Then j will be adjusted by $j = j + C_2[j] = 12 + 0 = 12$. It turns out that j at this point is the same as directly obtained from inverse BWT, because there is no duplication context for $j = 12$. Now with the newly updated j to be 12, and decremented i to be 11, we can retrieve the next character $S[i] = S[11] = L_2[j] = L_2[12] =$ "`i`". At this moment, the retrieved suffix of S is "`i$`".

Following the same updating sequence given by the loop of lines 11–16, if one traces the retrieval process, she will see that up to $i = 7$ backwardly from $i = 12$, every step behaves exactly like the inverse BWT as if C_2 is of no use at all because the visited values of C_2 are all initialized to be 1. But at the end of step $i = 7$, we see $j = T_2[j] = T_2[1] = 8$ and $C_2[8] = 2$, which correctly reflects there are 2 rows having the duplicated context. Hence, $C_2[8] = C_2[8] - 1 = 1$, $j = j + C_2[j] = 8 + 1 = 9$. This time, it differs from the previous steps in that the true j is clearly adjusted by the nonzero value of $C_2[8]$, which dynamically keeps track of how many times one context has not been encountered so far. As a result, the next character to be retrieved is $S[i] = S[6] = L_2[j] = L_2[9] =$ "`s`". At this moment, the retrieved suffix has been extended to "`ssippi$`". This procedure can go on until S is completely retrieved from L_2. The details of most steps are omitted here due to space concern. A more complete step-by-step runtime trace of our proposed inverse ST framework on an order-4 example can be found in our previous work [Nong and Zhang 2007b] for inverting ST in a time complexity of $O(N \log k)$.

As we can see, C_k and T_k suffice to restore a transformed text in a manner of linear-time backward retrieval. However, D_k turns out to be the most important vector in our proposed inverse ST framework. This is because once D_k is available, we can easily compute C_k and T_k in linear time from D_k (in the 3rd module). In order to retrieve S using a simple vector-lookup-based approach, we explored the following properties of M_k, where Property 3.6 directly comes from the definitions of M_k, L_k and T_k, and Property 3.7 is an immediate result due to the lexicographical sorting mechanism of ST. These two properties are utilized to build modules 3 and 4 in Figure 3.

PROPERTY 3.6. *For $k \in [2, N]$ and $i \in [1, N]$, $CT_k[T_k[i]] = CT_k[T_k[i] + 1] = \cdots = CT_k[T_k[i] + C_k[T_k[i]] - 1] = L_k[i]M_k[i, 1 : k - 1]$; for $k = 1$ and $i \in [1, N]$, $CT_k[T_k[i]] = CT_k[T_k[i] + 1] = \cdots = CT_k[T_k[i] + C_k[T_k[i]] - 1] = L_k[i]$.*

PROPERTY 3.7. *For $k \in [1, N]$, all the rows in M_k with the same first $h \in [1, k]$ characters are arranged consecutively.*

While modules 1, 3 and 4 require only $O(N)$ time and space complexity, the bottleneck of the whole framework resides in module 2: the complexity of computing D_k. We have previously shown that the best result for this module requires an $O(N \log k)$ time complexity [Nong and Zhang 2007b]. In the next section, we will present an even more efficient linear algorithm for computing D_k, which has a time and space complexity of $O(N)$, independent of k.

4. COMPUTING D_K IN $O(N)$ TIME AND SPACE

We first establish the direct mapping relationship between each entry of the switch vector D_k and the length of the Longest Common Prefixes (LCPs) of the corresponding two adjacent rows in CT_k. Then we introduce the definition of character cycle and show how to extract the cycles from L_k. Finally, by taking advantage of the properties of these cycles, we come up with a linear-time algorithm for computing D_k which leads to the linear-time inverse ST algorithm.

Let $lcp(i, j)$ denote the LCP between $CT_k[i]$ and $CT_k[j]$, where $i, j \in [1, N]$. Furthermore, we define $Height$ as a size-N vector, where $Height[i]$ stores the length of the LCP between the two k-order contexts of $CT_k[i-1]$ and $CT_k[i]$, that is, $Height[i] = \|lcp(i-1, i)\|$ for $i \in [2, N]$ and $Height[1] = 0$, where $\|\mathcal{X}\|$ denotes the length of string $\mathcal{X}$. For notation convenience, $Height[i]$ is also called the height of $L_k[i]$, that is, the height of the specific character at position i of L_k. From the definitions of D_k and $Height$, we see this property.

PROPERTY 4.1. *Given M_k, the following items regarding D_k and Height are equivalent.*

—$D_k[i] = 0$ *(as opposed to* $D_k[i] = 1$*);*
—$Height[i] = k$ *(as opposed to* $Height[i] \in [0, k-1]$*).*

It is obvious to see that once the vector $Height$ is available, D_k can be easily computed in $O(N)$ by traversing $Height$ once. Intuitively, this implies that we can convert the problem of constructing D_k to computing $Height$.

4.1. Character Cycles

Now, we introduce the definition of *character cycle*, or cycle in short, which builds the foundation for developing our algorithm to compute the vector D_k in linear time and space. Let $Q_k^x[i]$ denote the xth power of $Q_k[i]$, which is recursively defined as $Q_k^x[i] = Q_k[Q_k^{x-1}[i]]$ and $Q_k^0[i] = i$. For instance, $Q_k^3[i] = Q_k[Q_k[Q_k[i]]]$.

Definition 4.2. Cycle $\alpha(i)$ for $i \in [1, N]$, the list of characters consisting of a subset of L_k, is defined as $\alpha(i) = L_k[Q_k^0[i]]L_k[Q_k[i]]L_k[Q_k^2[i]]\ldots L_k[Q_k^m[i]]$, where m is the smallest nonnegative integer satisfying $Q_k^{m+1}[i] = i$. (Notice: since P_k and Q_k are reciprocal to each other, we can also use P_k instead of Q_k to define a cycle reversal to that defined on Q_k.)

In the preceding definition, $\|\alpha(i)\| = m+1$, hence $Q_k^{\|\alpha(i)\|}[i] = i$. In this sense, we term $\alpha(i)$ a cycle. Further, for any $L_k[j] \in \alpha(i)$ and $j \neq i$, we call $\alpha(i)$ and $\alpha(j)$ two *sibling* cycles.

Figure 4 gives an example for all the cycles in the 8-order ST of a sample text "`abababababababaab$`", where four disjoint (nonsibling) cycles are discovered. The rotated matrix M_8 is listed to facilitate the discussion here, but our algorithm will never restore the matrix, not even the 8-order contexts in the first 8 columns of M_8. The input is given at column L_8. The next three columns are F_8, P_8 and Q_8, which can be easily computed by traversing L_8 in linear time and space. To sort out a new cycle, we simply start with any unvisited row i, and follow $Q_8[i]$ to discover the next row. If the next row has not been visited before, we continue to follow $Q_8[Q_8[i]]$ and so on. This process will continue until we reach a previously visited row, which signifies that a complete cycle has been detected.

In the previous example, we will first visit $L_8[1]$ ("`b`"). Without loss of generality we name the cycle containing $L_8[1]$, C_1. To find the next character in C_1, we simply follow $Q_8[1]$. Because $Q_8[1]$ is 9, pointing to the 9th row, which has not been previously visited, we then pick the character at $L_8[9]$, which is "`a`". At this point, the first two characters in the C1 are "`ba`". Then again, we follow $Q_8[9]$ to find out the next row. This process will repeat until we reach the 10th row. Now, $Q_8[10]$ points to row 1, the row containing the first character of the current cycle, thus we close the current cycle. We say that all the visited characters form a cycle because no matter which character in the string is picked first, all the visited characters will be discovered in the same cyclic order by following Q_k. Hence, C_1 is "`baab$abababa`" of length 12. Similarly, using Q_8, we will continue to discover the other three length-2 cycles, C_2, C_3 and C_4, starting from $L_8[6]$, $L_8[7]$ and $L_8[8]$, respectively, that is, "`ba`", "`ba`", and "`ba`".

		Vectors				Cycles				Vectors		
M_8	R	L_8	F_8	P_8	Q_8	C_1	C_2	C_3	C_4	X_0	X_1	Y
aab$ababababababab	1	b	a	10	9	1				b	11	2
abaab$abababababab	2	b	a	11	10	11				a	10	12
ababaab$ababababab	3	b	a	12	11	9				a	9	10
abababaab$abababab	4	b	a	13	12	7				b	8	8
abababababababaab$	5	$	a	18	13	5				$	7	6
ababababababaab$ab	6	b	a	14	14		1			a	6	14
abababababaab$abab	7	b	a	15	15			1		b	5	16
ababababaab$ababab	8	b	a	16	16				1	a	4	18
ab$abababababababa	9	a	a	1	17	2				b	3	3
baab$ababababababa	10	a	b	2	1	12				a	2	1
babaab$abababababa	11	a	b	3	2	10				b	1	11
bababaab$ababababa	12	a	b	4	3	8				a	-11	9
bababababababaab$a	13	a	b	5	4	6				b	1	7
babababababaab$aba	14	a	b	6	6		2			a	-1	13
bababababaab$ababa	15	a	b	7	7			2		b	1	15
babababaab$abababa	16	a	b	8	8				2	a	-1	17
b$abababababababaa	17	a	b	9	18	3				b	1	4
$abababababababaab	18	b	$	17	5	4				a	-1	5

Fig. 4. A sample for four cycles.

Based on the preceding illustration, it is intuitive to see that any cycle whose size is not less than k has the associated context that can be immediately retrieved as the first k sequences of the cycle; while any cycle shorter gives rise to a periodic context that can be obtained by the repeated concatenation of the same short cycle.

4.2. Finding All the Cycles and Bookkeeping

We introduce three one-dimension size-N arrays X_0, X_1 and Y to store all the cycles obtained from L_k. These vectors will help retrieve any character in a context, which is necessary to compute the heights of characters in L_k. We proceed to show in what follows the algorithm for finding all the cycles from L_k in $O(N)$ time and space.

Initially, mark all the items of L_k as unvisited. Next, traverse L_k once from left to right to do as following. For each unvisited item $L_k[i]$, retrieve the cycle $\alpha(i)$ using Q_k in $O(\|\alpha(i)\|)$ time, and mark all the characters in this cycle as visited. All the characters of the found cycle $\alpha(i)$ are *consecutively* stored into X_0, where the two entries with the smallest and largest indices are called the *head* **and** *tail* of the cycle, respectively. For a cycle of length 1, both its head and tail point to the only entry. Notice that due to the cyclic property of a cycle, given $\alpha(i)$, we can retrieve any of its sibling cycles in a time complexity linear to the cycle's length. Therefore, all the sibling cycles of $\alpha(i)$ can share the space for $\alpha(i)$ in X_0, and they share the same head and tail in X_0. To help separate two neighbor nonsibling cycles stored in X_0, we maintain the head and tail positions of the characters belonging to each cycle by an array X_1. In X_1, we will assign to the entries corresponding to the tails of the cycles in X_0 nonpositive values, and the other entries' nonnegative values. Specifically, for a cycle of length $m > 1$, we assign $1 - m$ to the entry corresponding to its tail, which gives the distance of the head away from it. For the rest, the decreasing values $m - 1, m - 2, \ldots, 1$ are set to the entries from the head downwards, where the value of each entry gives the distance to the tail (from that entry). In particular, we assign 0 to the only entry in a cycle of length 1. To locate in X_0 the k-order context of each character $L_k[i]$, we use another array Y to map $CT_k[i][1]$ (i.e., the first character in the context of character $L_k[i]$, which is indeed $F_k[i]$ too) to its position in X_0, that is, $CT_k[i][1] = L_k[Q_k[i]] = X_0[Y[i]]$. An example of X_0, X_1

and Y is shown in Figure 4. From the description before, we see that extracting all the cycles from L_k and constructing the arrays X_0, X_1, and Y can be done in a total time and space complexity of $O(N)$.

From the definitions of head and tail of a cycle in X_0, and recalling $L_k[Q_k[i]] \in \alpha(i)$ (refer to Definition 4.2), we immediately see the following fact.

PROPERTY 4.3. *For any $i \in [1, N]$, the pair of sibling cycles $\alpha(i)$ and $\alpha(Q_k[i])$ share both the common head and tail in X_0.*

With X_0, X_1, and Y being defined in this way, some more parameters for $\alpha(i)$ can be obtained using the following formulas. First, the tail and head positions of the characters in cycle $\alpha(i)$ (as well as $\alpha(Q_k[i])$, refer to Property 4.3) stored in X_0 are given by $\mathcal{T}(i)$ and $\mathcal{H}(i)$ below respectively.

$$\mathcal{T}(i) = \begin{cases} Y[i] + X_1[Y[i]], & \text{for } X_1[Y[i]] \geq 0 \\ Y[i], & \text{otherwise} \end{cases}$$

$$\mathcal{H}(i) = \mathcal{T}(i) + X_1[\mathcal{T}(i)]$$

Next, the length of the cycle $\alpha(i)$, that is, $\|\alpha(i)\|$, is given by

$$\mathcal{L}(i) = \mathcal{T}(i) - \mathcal{H}(i) + 1.$$

In Figure 4, $L_8[1]$ and $L_8[2]$ are two characters in C_1, so $\alpha(1)$ and $\alpha(2)$ will see the same head and tail positions and cycle length in X_0 (refer to Property 4.3). That is, $\mathcal{H}(1) = \mathcal{H}(2) = 1$, $\mathcal{T}(1) = \mathcal{T}(2) = 12$ and $\mathcal{L}(1) = \mathcal{L}(2) = 12$. It is easy to see that all the three functions $\mathcal{T}(i)$, $\mathcal{H}(i)$ and $\mathcal{L}(i)$ are executed in $O(1)$ time. Now, we proceed to show how to utilize the developed facilities to retrieve any character in a context for the purpose of computing the context's height.

4.3. Retrieving Any Character in a Context

Similar to $Q_k^x[i]$, we define $P_k^x[i]$ to be the notation of the xth power of $P_k[i]$. From the definitions of CT_k, L_k, Q_k, P_k, and cycle, we observe the following property.

PROPERTY 4.4. *Given L_k and Q_k, we have*

(1) $CT_k[i] = L_k[Q_k[i]]L_k[Q_k^2[i]] \cdots L_k[Q_k^k[i]]$; *and*
(2) $Q_k^{\mathcal{L}(i)}[i] = P_k^{\mathcal{L}(i)}[i] = i$.

Further, for any given character in L_k, we observe the following property describing the relationship between the context of the character and the cycle containing the character.

PROPERTY 4.5. *The k-order context of $L_k[i]$ is the first k characters of the string made up of repetitions of cycle $\alpha(Q_k[i])$.*

PROOF. According to the definition of cycle given by Definition 4.2, we have $\alpha(Q_k[i]) = L_k[Q_k[i]]L_k[Q_k^2[i]] \cdots L_k[Q_k^{\|\alpha(Q_k[i])\|}[i]]$ and $Q_k^{\|\alpha(Q_k[i])\|+1}[i] = Q_k[i]$. Further, from Property 4.4, we have $CT_k[i] = L_k[Q_k[i]]L_k[Q_k^2[i]] \cdots L_k[Q_k^k[i]]$. By comparing $CT_k[i]$ and $\alpha(Q_k[i])$, we can immediately see that: (i) when $\|\alpha(Q_k[i])\| \geq k$, $CT_k[i]$ is the first k characters of $\alpha(Q_k[i])$; (ii) when $\|\alpha(Q_k[i])\| < k$, the cycle $\alpha(Q_k[i])$ will repeat itself in CT_k at each position $j \in [1, k]$ satisfying $(j-1)(\text{mod } \|\alpha(Q_k[i])\|) = 0$. □

After all the cycles have been extracted from L_k and recorded in the three arrays X_0, X_1, and Y, retrieving the character in any cyclic distance (not longer than $k-1$) from the character $X_0[Y[i]]$ can be done in time $O(1)$ using the following formula. That is, in

```
GetHeights(i)
1  h ← 0; j ← i;
2  for m ← 1 to ℒ(i)   ▷ Walk through the cycle α(i).
3      do
4         while h < k   ▷ k-order LCP is of length at most k.
5             do
6                if j = 1 or 𝒞(j, h) ≠ 𝒞(j − 1, h)
7                  then break;
8                h ← h + 1;
9         Height[j] ← h;
          ▷ Decrease h to compute the height of the succeeding char.
10        if h > 0
11          then h ← h − 1;
12        j ← Q_k[j];   ▷ Move to the index of the succeeding char.

13 return Height;
```

Fig. 5. The algorithms for computing the heights of all the characters in $\alpha(i)$.

the k-order context of character $L_k[i]$, any character $CT_k[i][d+1]$, $d \in [0, k-1]$ (notice: the indices of items in CT_k start from 1), is given by

$$\mathcal{C}(i, d) = X_0[(Y[i] - \mathcal{H}(i) + d)(\text{mod } \mathcal{L}(i)) + \mathcal{H}(i)].$$

4.4. Computing the Heights for a Cycle

In the previous subsections, we have introduced how to determine the value of $D_k[i]$ by the height of $L_k[i]$, and stated that the construction of D_k is the final challenge for building the linear inverse ST algorithm. To achieve this goal, we are going to establish a linear algorithm for computing the heights of all the characters in a cycle. The following lemma tells us the inductive relation between the heights of consecutive characters in a cycle, which establishes in the context of ST the result of a folklore in the string matching community [Kasai et al. 2001].

LEMMA 4.6. *For* $k \in [1, N]$, $i \in [2, N]$ *and* $Height[i] \geq 1$, *there must be* $Height[Q_k[i]] \geq Height[i] - 1$.

PROOF. Suppose $Height[i] = \|lcp(i-1, i)\| = l \geq 1$. When $l = 1$, the statement is correct since we always have $Height[Q_k[i]] \geq 0$. Now, let's consider $l > 1$. According to the definition of Q_k, we have $Q_k[i-1] < Q_k[i]$, and the $(l-1)$-order contexts of all characters $L_k[j]$, $j \in [Q_k[i-1], Q_k[i]]$, must be the same. Hence, $Height[Q_k[i]] \geq l - 1 = Height[i] - 1$. □

With Lemma 4.6, the heights of all the characters in a single cycle $\alpha(i)$ can be retrieved consecutively one-by-one by the algorithm `GetHeights`[3] presented in Figure 5. In that algorithm, the height of each individual character $L_k[j]$ in $\alpha(i)$ is computed by the loop of lines 4–8, in which the two characters from the two neighboring contexts $CT_k[j]$ and $CT_k[j-1]$, currently being compared, are retrieved by $\mathcal{C}(j, h)$ and $\mathcal{C}(j-1, h)$ in $O(1)$ time, respectively.

Further, let $\beta(i) = \{L_k[j+1] \mid j \in [1, N-1] \text{ and } L_k[j] \in \alpha(i)\}$ be the set of all the right-hand neighbors (in L_k) of the characters in $\alpha(i)$, which we call the **cushion** of $\alpha(i)$. We have another algorithm `GetHeights1`, which is similar to `GetHeights`, for computing

[3]This algorithm is similar to GetHeight in Kasai et al. [2001] for computing the LCP of BWT, but differs in that the former computes the heights for all the characters of a single cycle in L_k, where $k \in [1, N]$, however, the latter computes the heights of all the characters in L_N.

the heights of all the characters in $\beta(i)$. The algorithm `GetHeights1` is simply derived from `GetHeights` by revising *only* lines 6 and 9 as following:

—Line 6: **if** $j = N$ or $\mathcal{C}(j, h) \neq \mathcal{C}(j + 1, h)$
—Line 9: **if** $j < N$ **then** $Height[j + 1] \leftarrow h$;

Hence, for each $L_k[j] \in \alpha(i)$, `GetHeights`(i) gets the height of it by computing $lcp(j - 1, j)$, while `GetHeights1`(i) gets the height of its right-hand neighbor, $L_k[j + 1]$, by computing $lcp(j, j+1)$ in a symmetric manner (i.e., replacing $j-1$ and j by j and $j+1$, respectively).

We now analyze the time complexities of `GetHeights` and `GetHeights1` by induction. Let $T_{inc}(m)$ and $T_{dec}(m)$ be the cumulative execution times of lines 8 and 11 in the first m iteration(s) of the outer loop, respectively, and $\Delta T(m) = T_{inc}(m) - T_{dec}(m)$. Initially, we have $\Delta T(1) \leq k-1$. Now, suppose we see the first time $\Delta T(m_1) = k-1$ for some $m_1 \geq 1$, line 8 can execute at most once in the (m_1+1)th iteration. Given line 8 executes, line 11 must execute once too. Hence, $\Delta T(m_1 + 1)$ must be nonincreasing from $\Delta T(m_1)$, that is, $\Delta T(m_1+1) \leq \Delta(m_1) = k-1$. In conclusion, we have $\Delta T(m) \leq k-1$ for $m \in [1, \mathcal{L}(i)]$. Since $T_{dec}(\mathcal{L}(i)) \leq \mathcal{L}(i)$, we further have $T_{inc}(\mathcal{L}(i)) \leq \mathcal{L}(i)+k-1$. Therefore, the time complexity of `GetHeights` is $O(T_{inc}(\mathcal{L}(i)) + T_{dec}(\mathcal{L}(i))) = O(k + \mathcal{L}(i))$. The preceding same analysis can be applied to `GetHeights1`. As a result, the total time complexity for `GetHeights` and `GetHeights1` is $O(k+\mathcal{L}(i))$, which suggests the following lemma regarding the time complexity for obtaining the heights of characters in $\alpha(i)$ and its cushion $\beta(i)$.

LEMMA 4.7. *For any cycle $\alpha(i)$, $i \in [1, N]$, the heights of all characters in $\alpha(i) \cup \beta(i)$ can be computed in a time complexity of $O(k + \mathcal{L}(i))$.*

4.5. Computing D_k

Let $CH_0 = \{L_k[i] \mid i \in [1, N]$ and $\mathcal{L}(i) > k/2\}$, $CH_1 = \{L_k[i+1] \mid i \in [1, N-1]$ and $\mathcal{L}(i) > k/2\}$, and $CH_2 = L_k - (CH_0 \cup CH_1)$. Given these definitions, CH_2 can be nonempty only when $k \geq 2$. Further, for any $L_k[i] \in CH_2$, we have that (i) $\mathcal{L}(i) \leq k/2$ when $i = 1$; and (ii) $\mathcal{L}(i) \leq k/2$ and $\mathcal{L}(i - 1) \leq k/2$ when $i \in [2, N]$.

We first give in Figure 6 the algorithm `MakeDk` for computing D_k in linear time $O(N)$. In this algorithm, lines 1–3 are done in $O(N)$ time, as we have explained before. For the loop of lines 4–8, line 8 is executed to compute the heights for characters in $\alpha(i) \cup \beta(i)$, where $\alpha(i)$ is longer than $k/2$. From Lemma 4.7, we see that each running of line 8 will have a time complexity of $O(k + \mathcal{L}(i)) = O(\mathcal{L}(i))$, for the sake of $\mathcal{L}(i) > k/2$. For any $Height[j]$, $j > i$, computed by line 8, its value is nonnegative and will be filtered out by line 6 when i is increased to scan L_k from left to right. Hence, conditioned by line 6, the time complexity of this loop is $O(N)$. Following the first loop, the second loop in lines 9–17 computes each $D_k[i]$ for $L_k[i] \in CH_0 \cup CH_1$ by inspecting the nonnegative $Height[i]$ obtained so far, which is done by simply scanning $Height$ from left to right. The time complexity of this loop is obviously seen to be $O(N)$. In the third loop, D_k is scanned from left to right to compute each $D_k[i]$ for $L_k[i] \in CH_2$, that is, to finish the computing of D_k. Conditioned by line 20, lines 22–30 are executed only when $L_k[i] \in CH_2$. In lines 28 and 30, calling `Propagate`(i, v) will walk through $\alpha(i)$ to set $D_k[j] = v$ for each unassigned $D_k[j]$, where $L_k[j] \in \alpha(i)$; for any $j > i$, $D_k[j]$ will be filtered out by line 20 when i is increased to complete the loop (of lines 18–30). Therefore, the 3rd loop is done in time of $O(N)$. As a result, we conclude that `MakeDk` has a total time complexity of $O(N)$.

Now, we prove the correctness of `MakeDk`. The correctness of the first two loops is directly ascertained by the algorithms `GetHeights` and `GetHeights1`. Further, we are going to establish some results in the following to assure that the 3rd loop is correct. In the rest of this section, we say $\alpha(i) = \alpha(i - 1)$ if $\mathcal{L}(i) = \mathcal{L}(i - 1)$ and $\alpha(i)[j] = \alpha(i - 1)[j]$

```
MakeDk(L_k, N)
Compute P_k and Q_k from L_k;
Find all the cycles from L_k using P_k and Q_k, and record in X_0, X_1 and Y;
Initialize Height and D_k by -1;

▷ Compute Height[i] for each L_k[i] ∈ CH_0 ∪ CH_1.
for i ← 1 to N
      do
         if Height[i] < 0 and ℒ(i) > k/2
            then
                  GetHeights(i); GetHeights1(i);

▷ Compute D_k[i] for each L_k[i] ∈ CH_0 ∪ CH_1.
for i ← 1 to N
      do
         if Height[i] > −1
            then
                  if Height[i] < k
                     then
                           D_k[i] ← 1;
                     else
                           D_k[i] ← 0;

▷ Compute D_k[i] for each L_k[i] ∈ CH_2.
for i ← 1 to N
      do
         if D_k[i] < 0
            then
                  if i = 1 or ℒ(i) ≠ ℒ(i − 1)
                     then
                           D_k[i] = 1;
                     else
                           if α(i) = α(i − 1)   ▷ The comparison is done in time of O(ℒ(i)).
                              then
                                    Propagate(i, 0);
                              else
                                    Propagate(i, 1);

return D_k;

Propagate(i, v)
▷ Walk through α(i) to set D_k[j] = v for each unassigned D_k[j].
j ← i;
for m ← 1 to ℒ(i)
      do
         if D_k[j] < 0
            then
                  D_k[j] = v;
         j = Q_k[j];
```

Fig. 6. The linear algorithm for computing D_k.

(i.e., two equivalent characters) for any $j \in [1, \mathcal{L}(i)]$, and $\alpha(i) \neq \alpha(i-1)$ if $\alpha(i) = \alpha(i-1)$ is not true.

Lemma 4.8. *For any $L_k[i] \in CH_2$, $i \in [2, N]$, we have $CT_k[i] = CT_k[i-1]$ if and only if $\alpha(i) = \alpha(i-1)$.*

PROOF. The sufficiency comes directly from Property 4.5. Now, we turn to the necessity. Given $L_k[i] \in CH_2$, let $l_i = \mathcal{L}(i)$ and $l_{i1} = \mathcal{L}(i-1)$. Without loss of generality we suppose $l_{i1} \geq l_i$. Furthermore, from the definition of CH_2, $l_i \leq k/2$ and $l_{i1} \leq k/2$. The proof is conducted in two steps that follow.

(1) We first prove that $L_k[i] = L_k[i-1]$. From the cycle property, given $l_{i1} \leq k/2$, we know that (i) $CT_k[i-1][l_{i1}] = L_k[i-1]$. Again from the cycle property and under the assumption $l_{i1} \geq l_i$, we have $CT_k[i][1:l_i] = CT_k[i-1][1:l_i] = CT_k[i-1][l_{i1}+1:l_{i1}+l_i] = CT_k[i][l_{i1}+1:l_{i1}+l_i]$, that is, (ii) $CT_k[i][1:l_i] = CT_k[i][l_{i1}+1:l_{i1}+l_i]$. Because $CT_k[i][l_{i1}+1:l_{i1}+l_i]$ is a cycle, we have (iii) $CT_k[i][l_{i1}+1-l_i:l_{i1}] = CT_k[i][l_{i1}+1:l_{i1}+l_i]$. Combining (ii) and (iii), we get $CT_k[i][1:l_i] = CT_k[i][l_{i1}+1-l_i:l_{i1}]$, which plus $CT_k[i][l_i] = L_k[i]$ yields (iv) $CT_k[i][l_{i1}] = L_k[i]$. Now we recall the assumption $CT_k[i] = CT_k[i-1]$ to obtain (v) $CT_k[i][l_{i1}] = CT_k[i-1][l_{i1}]$. With (i), (iv) and (v), we see that $L_k[i] = L_k[i-1]$.

(2) Next, we utilize $L_k[i] = L_k[i-1]$ to complete the proof. Let $j = P_k[i]$ and $j_1 = P_k[i-1]$. Because that $CT_k[j] = L_k[i]CT_k[i][1:k-1]$ and $CT_k[j_1] = L_k[i-1]CT_k[i-1][1:k-1]$, we have $CT_k[j] = CT_k[j_1]$. Further, $L_k[j]$ and $L_k[i]$ are in the same cycle $\alpha(i)$, and $L_k[j_1]$ and $L_k[i-1]$ are in the same cycle $\alpha(i-1)$ too. This suggests that $\mathcal{L}(i) = \mathcal{L}(j)$ and $\mathcal{L}(i-1) = \mathcal{L}(j_1)$. Hence $L_k[j] \in CH_2$. Given $L_k[i] = L_k[i-1]$, we have, from the definition of P_k, that $j_1 = j-1$. Thus, we may apply the analysis in (1) for rows i and $i-1$ to rows j and $j-1$ again to show $L_k[j] = L_k[j-1]$. By repeating the process $\mathcal{L}(i)$ times, we see that $\alpha(i) = \alpha(i-1)$. □

COROLLARY 4.9. *For any $L_k[i] \in CH_2$, $i \in [2, N]$, if $CT_k[i] = CT_k[i-1]$, we must have $CT_k[j] = CT_k[j-1]$ for any character $L_k[j] \in \alpha(i)$.*

PROOF. As we have shown in (1) of the proof for Lemma 4.8, we have $L_k[i] = L_k[i-1]$. Further, notice that $CT_k[P_k[i-1]] = L_k[i-1]CT_k[i-1][1:k-1]$ and $CT_k[P_k[i]] = L_k[i]CT_k[i][1:k-1]$, we have $CT_k[P_k[i-1]] = CT_k[P_k[i]]$. From (2) of the proof for Lemma 4.8, we know that $P_k[i-1] = P_k[i]-1$. Hence we have $CT_k[P_k[i]-1] = CT_k[P_k[i]]$. Repeating this analysis for $j = P_k^m[i]$, $m \in [1, \mathcal{L}(i)]$, we have $CT_k[j] = CT_k[j-1]$ for each $L_k[j] \in \alpha(i)$. □

COROLLARY 4.10. *For any $L_k[i] \in CH_2$, $i \in [2, N]$, if $CT_k[i] \neq CT_k[i-1]$, we must have $CT_k[j] \neq CT_k[j-1]$ for any character $L_k[j] \in \alpha(i)$ and $L_k[j] \in CH_2$.*

PROOF. We prove it by contradiction. Suppose that there is $CT_k[j] = CT_k[j-1]$ for any $L_k[j] \in \alpha(i)$ and $L_k[j] \in CH_2$. Notice that $CT_k[i][1]$ is in $\alpha(i)$ as well as $\alpha(j)$, from Corollary 4.9, we have $CT_k[i] = CT_k[i-1]$. This contradicts the assumption. □

Given these results, we now return to prove the correctness of the 3rd loop in `MakeDk`. Notice that when line 22 is reached, $L_k[i]$ must be in CH_2. At this point, we first check whether $\mathcal{L}(i) = \mathcal{L}(i-1)$ in time $O(1)$, that is, to test if the two cycles have the same length, where $i = 1$ in the condition is to cover $D_k[1]$ which is always of value 1 by definition. If no, we can immediately determine $\alpha(i) \neq \alpha(i-1)$ as well as $CT_k[i] \neq CT_k[i-1]$ (refer to Lemma 4.8). Otherwise, we further compare $\alpha(i)$ and $\alpha(i-1)$ in line 22 to determine if $CT_k[i] = CT_k[i-1]$ (refer to Lemma 4.8), which can be done by checking $\mathcal{C}(i, d)$ with $\mathcal{C}(i-1, d)$ for $d \in [0, \mathcal{L}(i)-1]$ in at most $O(\mathcal{L}(i))$ time. Then we can walk through $\alpha(i)$ to propagate the result to each $D_k[j] \in \alpha(i)$ which is unassigned so far (refer to Corollary 4.9 and 4.10). As a result, the 3rd loop will correctly compute the value of $D_k[i]$ for any $L_k[i] \in CH_2$.

Summarizing the preceding analysis, we have the following theorem to state the linearity of our algorithm for computing D_k as well as the inverse ST for any L_k.

THEOREM 4.11 (LINEAR TIME/SPACE). *Given L_k, we can restore the original text of S in $O(N)$ time and space for any $k \in [1, N]$.*

5. CONCLUSION

The inverse ST solutions proposed in Schindler [1997, 2001] and Yokoo [1999] rely on the complete restoration of all the k-order contexts and employ hash tables to look up the context for every character being restored. Such inverse ST approaches differ significantly from the way how the inverse BWT works. In our previous works [Nong and Zhang 2007b], we have proposed a new algorithm framework that uses the size-N auxiliary vectors D_k, T_k, and C_k, instead of M_k, to retrieve the original text from L_k, thus requiring a space of $O(N)$ only. However, the best time complexity obtained before this article was $O(N \log k)$. In this article, we present a linear-time solution for the inverse ST problem, which further advances the research on this topic. It is worth noting that the method we present here for computing D_k (which is the bottleneck of the whole algorithm) in linear time is completely different from that in our previous works for achieving $O(kN)$ and $O(N \log k)$ time. In this sense, not only is the linear-time result new, but also is the method we used to achieve the result novel. For sample implementations, all the codes for our inverse ST algorithms are available at this URL: `http://www.cs.sysu.edu.cn/nong/pub/ist/ist_alg.zip`.

ACKNOWLEDGMENT

The authors wish to express special thanks to the TALG reviewers and editor for this article, the reviewers for the simplified version for CPM'08 [Nong et al. 2008], and Professor S. Chiu in the Mathematics Department of Hong Kong Baptist University, for their consistent efforts on contributing constructive suggestions and insightful comments for improving the presentation of this article.

REFERENCES

ADJEROH, D., BELL, T., AND MUKHERJEE, A. 2008. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.

ADJEROH, D., ZHANG, Y., MUKHERJEE, A., POWELL, M., AND BELL, T. 2002. DNA sequence compression using the Burrows-Wheeler transform. In *Proceedings IEEE Computer Society Bioinformatics Conference*. 303–13.

ARNAVUT, Z. 1999. Lossless compression of color-mapped images. *Optic. Engin. 38,* 6, 1001–5.

ARNAVUT, Z. 2001. Lossless and near-lossless compression of ECG signals. In *Conference Proceedings of the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Vol. 3. 2146–2149.

BALKENHOL, B. AND KURTZ, S. 2000. Universal data compression based on the Burrows-Wheeler transformation: Theory and practice. *IEEE Trans. Comput. 49,* 10, 1043–53.

BALKENHOL, B., KURTZ, S., AND SHTARKOV, Y. M. 1999. Modifications of the Burrows and Wheeler data compression algorithm. In *Proceedings of the Data Compression Conference (DCC'99)*. 188–97.

BARON, D. AND BRESLER, Y. 2005. Antisequential suffix sorting for BWT-based data compression. *IEEE Trans. Comput. 54,* 4, 385–97.

BIRD, R. S. AND MU, S. C. 2004. Inverting the burrows-wheeler transform. *J. Funct. Program. 14,* 6, 603–612.

BURROWS, M. AND WHEELER, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. rep. SRC res. rep. 124, Digital Systems Research Center, CA. May.

DEOROWICZ, S. 2002. Second step algorithms in the Burrows-Wheeler compression algorithm. *Softw. Pract. Exper. 32,* 2, 99–111.

GROSSI, R. AND VITTER, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput. 35,* 2, 378–407.

KARKKAINEN, J. AND SANDERS, P. 2003. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*. 943–955.

KASAI, T., LEE, G., ARIMURA, H., ARIKAWA, S., AND PARK, K. 2001. Linear-Time longest-common-prefix computation in suffix arrays and its applications. In *Lecture Notes in Computer Science vol. 2089*, 181–192.

KIM, D. K., SIM, J. S., PARK, H., AND PARK, K. 2003. Linear-Time construction of suffix arrays. In *Proceedings 14th Annual Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 2676. Springer, 186–199.

KO, P. AND ALURU, S. 2003. Space efficient linear time construction of suffix arrays. In *Proceedings 14th Annual Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 2676. Springer, 200–210.

LARSSON, N. AND K.SADAKANE. 1999. Faster suffix sorting. Tech. rep. LU_CS-TR-99-214, Department of Computer Science, Lund University, Sweden.

MANZINI, G. 1999. The Burrows-Wheeler transform: theory and practice. In *Proceedings of the 24th International Symposium.* Lecture Notes in Computer Science, vol. 1672. 34–47.

MANZINI, G. AND FERRAGINA, P. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40,* 1.

MUKHERJEE, A., MOTGI, N., BECKER, J., FRIEBE, A., HABERMAN, C., AND GLESNER, M. 2001. Prototyping of efficient hardware algorithms for data compression in future communication systems. In *Proceedings of the 12th International Workshop on Rapid System Prototyping (RSP'01)*. 58–63.

NONG, G. AND ZHANG, S. 2006. Unifying the Burrows-Wheeler and the Schindler transforms. In *Proceedings of the Data Compression Conference (DCC'06)*. 464.

NONG, G. AND ZHANG, S. 2007a. An efficient algorithm for the inverse ST problem. In *Proceedings of the Data Compression Conference.* 397.

NONG, G. AND ZHANG, S. 2007b. Efficient algorithms for the inverse sort transform. *IEEE Trans. Comput. 56,* 11, 1564–74.

NONG, G., ZHANG, S., AND CHAN, W. H. 2008. Computing inverse ST in linear complexity. In *Proceedings of the 19th Combinatorial Pattern Matching Conference*. 178–190.

PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. 2005. The performance of linear time suffix sorting algorithms. In *Proceedings of the Data Compression Conference (DCC'05)*. 358–367.

SCHINDLER, M. 1997. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the Data Compression Conference (DCC'97)*. 469.

SCHINDLER, M. 1999. Szip homepage. http://www.compressconsult.com/szip/.

SCHINDLER, M. 2000. The sort transformation. http://www.compressconsult.com/st/.

SCHINDLER, M. 2001. Method and apparatus for sorting data blocks. US Patent 6199064.

SCHINDLER, M. AND SEBASTIAN, B. 2001. Image compression using blocksort. In *Proceedings of the Data Compression Conference (DCC'01)*. 515.

YOKOO, H. 1999. Notes on block-sorting data compression. *Electron. Comm. Japan Part III: Fundam. Electron. Sci. 82,* 6, 18–25.

Received November 2007; revised September 2010; accepted September 2010