

# Computing Inverse ST in Linear Complexity

Ge Nong<sup>1\*</sup>, Sen Zhang<sup>2</sup>, and Wai Hong Chan<sup>3\*\*</sup>

<sup>1</sup> Computer Science Department, Sun Yat-Sen University, P.R.C.,  
`issng@mail.sysu.edu.cn`

<sup>2</sup> Dept. of Math., Comp. Sci. and Stat., SUNY College at Oneonta, U.S.A.,  
`zhangs@oneonta.edu`

<sup>3</sup> Department of Mathematics, Hong Kong Baptist University, Hong Kong,  
`dchan@hkbu.edu.hk`

**Abstract.** The Sort Transform (ST) can significantly speed up the block sorting phase of the Burrows-Wheeler transform (BWT) by sorting only 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 paper, we present a novel algorithm that can compute the inverse ST in an  $O(N)$  time/space complexity, a linear result independent of  $k$ . The main idea behind the design of the linear algorithm is a set of cycle properties of  $k$ -order contexts we explored 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, leading to the proposed linear inverse ST algorithm.

## 1 Introduction

Since the Burrows-Wheeler transform (BWT) was introduced in 1994 [1], it has been successfully used in a wide range of data compression applications. 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 [2, 3], which can speed up the block sorting phase of the BWT by sorting only a portion of the rotating matrix. 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 rotating 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 rotating matrix, i.e., the sorting

---

\* Nong was partially supported by the National Natural Science Foundation of P.R.C. (Project No. 60573039).

\*\* Chan was partially supported by the Faculty Research Grant (FRG/06-07/II-28), Hong Kong Baptist University and the CERG (HKBU210207), RGC, Hong Kong SAR.

in ST is stable. 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 retaining high compression ratios. Schindler has built a fast compression software called *szip* [2] using the ST.

## Problem

A major tradeoff caused by the ST’s partial sorting scheme is that the inverse ST is more complex than the inverse BWT. This is because although each of the 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 in which 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 [4], resulting in an  $O(kN)$  time/space complexity for inverting the ST of a size- $N$  string. 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 [5–7], which is similar to that used for the inverse BWT [1], but different from any possible hash table based approaches suggested by Schindler [4], Yokoo [8] and Bird [9]. This framework requires only  $O(N)$  space complexity, however, the time complexity of our previous solutions remains to be superlinear. The time complexity achieved in [5] is  $O(kN)$ , which we have recently reduced to  $O(N \log k)$  in [6, 7]. Nevertheless, 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/space complexity of  $O(N)$ . Therefore, the question of our particular interest here is whether the inverse ST is linear computable.

## Answer

We present here a positive answer to this question by introducing 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/space complexity  $O(N)$ , independent of  $k$ .

Section 2 introduces some basic definitions and general notations. Our linear inverse ST algorithm is developed and analyzed in section 3.

## 2 Preliminary

A text  $S$  of length  $N$  is denoted as  $x_1x_2x_3\dots x_{N-1}\$$ , where each character  $x_i \in \Sigma$ ,  $i \in [1, N-1]$  and  $\Sigma$  is the alphabet. The last character  $\$$  of the text is a *sentinel*, which is the unique lexicographically greatest <sup>4</sup> character in  $S$ . (Appending a

---

<sup>4</sup> Symmetrically, the sentinel can be assumed as the smallest.

sentinel to the original text has been used in many previous publications; readers may refer to papers [10, 11] for more details.) Given  $S$ , according to the cyclic rotation scheme, 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  $S[1]$ . Cyclic rotating  $S$  a total number of  $N$  times, we obtain the original matrix  $M_0$  for computing the ST of  $S$ .

**Definition 1.** (*The original matrix  $M_0$* ). The  $N \times N$  symmetric matrix originally constructed from the texts obtained by rotating the text  $S$ . Specifically, the first row of the matrix  $M_0$  is assigned to be  $S$ , 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.

Each row in  $M_0$  is a text, where the first  $k$  characters is called the  $k$ -order context of the last character, for  $k \in [1, N]$ . When  $k = N$ , the  $k$ -order context is also called the unlimited order context; or else a limited order context. Lexicographically sorting all the rows of  $M_0$ , we get a new matrix  $M_k$ , which last column is the ST of  $S$ . For an example of the ST, we give in Fig. 1 the matrix  $M_2$  for  $S = [m i s s i s s i p p i \$]$ . 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 transform result can be denoted as a couplet ( $[s m s p \$ p i s s i i], 5$ ).

$$\begin{bmatrix} i & p & p & i & \$ & m & i & s & s & i & s & s \\ i & s & s & i & s & s & i & p & p & i & \$ & m \\ i & s & s & i & p & p & i & \$ & m & i & s & s \\ i & \$ & m & i & s & s & i & s & s & i & p & p \\ m & i & s & s & i & s & s & i & p & p & i & \$ \\ p & i & \$ & m & i & s & s & i & s & s & i & p \\ p & p & i & \$ & m & i & s & s & i & s & s & i \\ s & i & s & s & i & p & p & i & \$ & m & i & s \\ s & i & p & p & i & \$ & m & i & s & s & i & s \\ s & s & i & s & s & i & p & p & i & \$ & m & i \\ s & s & i & p & p & i & \$ & m & i & s & s & i \\ \$ & m & i & s & s & i & s & s & i & p & p & i \end{bmatrix}$$

**Fig. 1.** The matrix  $M_2$  for the 2-order ST, where the last column is the transformed text.

For presentation simplicity, we introduce the following notations. Let  $Z[N_r, N_c]$  represent a two-dimensional array  $Z$  consisting of  $N_r$  rows and  $N_c$  columns. To specify an array's subscript range in each dimension, we use the notation of  $a : b$ . For example,  $Z[a : b, c : d]$  represents a 2-D sub-array of  $Z[N_r, N_c]$  covering 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 case  $a = b$  and/or  $c = d$ , the simpler forms of  $Z[a, c : d]$  or  $Z[a : b, c]$  are used instead, respectively. From  $M_k$ , we define  $F_k = M_k[1 : N, 1]^T$

and  $L_k = M_k[1 : N, N]^T$ , i.e. the transposes of the first and the last columns, respectively, where  $k \in [1, N]$ . When  $k = N$ , the simpler forms of  $M$ ,  $F$  and  $L$  can be used for  $M_k$ ,  $F_k$  and  $L_k$  instead, respectively.

```

IBWT(char *L, int start, int n) {
// restore n characters starting from L[i]
j=start;
for(i=n; i>0; i--) {
  S[i]=L[j]; // restore the ith character.
  j=P[j]; // update index for backward retrieving the preceding character.
}
return S;
}

```

**Fig. 2.** Algorithm for the inverse BWT.

### 3 The Linear Inverse ST Algorithm

#### 3.1 Basis

Let's define two vectors to establish an one-to-one mapping among the characters of  $F_k$  and  $L_k$ , as below.

**Definition 2.** ( $P_k$  and  $Q_k$ ).  $P_k$  and  $Q_k$  are two size- $N$  row vectors, where the former satisfies

$$\begin{cases} F_k[P_k[i]] = L_k[i], & \text{for } i \in [1, N]; \\ P_k[i] < P_k[j], & \text{for } (1 \leq i < j \leq N \text{ and } L_k[i] = L_k[j]) \text{ or } (L_k[i] < L_k[j]). \end{cases}$$

and the later satisfies

$$\begin{cases} L_k[Q_k[i]] = F_k[i], & \text{for } i \in [1, N]; \\ Q_k[i] < Q_k[j], & \text{for } 1 \leq i < j \leq N \text{ and } F_k[i] = F_k[j]. \end{cases}$$

$P_k$  maps the index of each character of  $L_k$  to its index at  $F_k$ , and  $Q_k$  maps the index of each character at  $F_k$  to its index at  $L_k$ . Furthermore,  $P_k$  and  $Q_k$  are reciprocal to each other, i.e.  $Q_k[P_k[i]] = i$  and  $P_k[Q_k[i]] = i$ . When  $k = N$ , the simpler forms of  $P$  and  $Q$  can be used for  $P_k$  and  $Q_k$  instead. The solution for the inverse BWT [1] is re-stated in Fig. 2, where  $S$  is restored by *backward retrieval* using the vector  $P$  to iteratively retrieve the immediate preceding character  $S[i - 1]$  for each known  $S[i]$ , utilizing the following properties.

*Property 1.* Given  $L$ ,  $F$  can be obtained by sorting all the characters of  $L$ .

*Property 2.* In both  $F$  and  $L$ , the relative orders of any two identical characters are consistent.

*Property 3.* Given  $L[i] = S[j]$ , we have  $S[j - 1] = L[P[i]]$  for  $j \in [2, N]$ , and  $S[N] = L[P[i]]$  for  $j = 1$ .

### 3.2 Algorithm Framework

We previously proposed in [5, 7] an auxiliary vector based framework to inverse ST using no hashing table. The most complex part in that framework is computing the  $k$ -order context switch vector  $D$ , which is defined as following.

To denote the  $k$ -order contexts in  $M_k$ , we define the  $k$ -order context vector  $CT_k$ , which is a size- $kN$  vector with each  $CT_k[i]$  denoting the  $k$ -order context of  $L_k[i]$ , i.e.  $CT_k[i] = M_k[i, 1 : k]$ , where  $i \in [1, N]$ . From  $CT_k$ , we further define the  $k$ -order context switch vector  $D$  as below.

**Definition 3.** ( *$k$ -order context switch vector  $D$* ). A size- $N$  row vector with each  $D[i]$ ,  $i \in [1, N]$  defined as

$$D[i] = \begin{cases} 0, & \text{for } CT_k[i] = CT_k[i-1]; \\ 1, & \text{for } CT_k[i] \neq CT_k[i-1]. \end{cases}$$

For  $CT_k$ , we say there is a  $k$ -order context switch from row  $i-1$  to row  $i$  if either it is the first context (when  $i=1$ ) or it is different from that at the  $(i-1)$ th row (for  $i \in [2, N]$ ). Suppose that  $D$  has been known, we need the following two size- $N$  vectors  $C_k$  and  $T_k$  to use with  $D$  together to restore  $S$  from  $L_k$ , in a fashion of backward retrieval.  $C_k$  is called the counter vector, which is a size- $N$  vector recording the occurrence of each unique  $k$ -order context at its first position in the  $M_k$ . If  $C_k[i] > 0$ , the  $k$ -order context at the  $i$ th row is new and is repeated for  $C_k[i]$  times starting from the  $i$ th row consecutively up to the  $(i + C_k[i] - 1)$ th row; otherwise, the  $k$ -order context at the  $i$ th row repeats the one at the previous row.  $T_k$  is called the index vector, which is a size- $N$  vector pointing to the starting row of each unique  $k$ -order context.  $M_k[T_k[i], 2 : k]$  is the  $(k-1)$ -order context of the character  $L_k[i]$ , where  $i \in [1, N]$ . 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 with  $L_k[i]$  being the first character.

The complexity of calculating  $D$  constitutes the bottleneck of the whole framework. We have previously shown the best result for computing  $D$  from  $L$  requires an  $O(N \log k)$  time complexity [7] and using a space  $O(N)$ . In the next subsection, we will present an even more efficient linear algorithm, which has a time/space complexity of  $O(N)$ , independent of the context order  $k$ .

### 3.3 Computing $D$ in $O(N)$ time/space

We first show the relationship between the vector  $D$  and the lengths of the longest common prefixes (LCPs) between any two adjacent rows in  $M_k[1 : N, 1 : k]$ , then present a linear algorithm that can compute  $D$  in a  $k$ -independent time/space complexity of  $O(N)$  by exploring the cycle and the LCP properties. Let  $lcp(i, j)$  denote the longest common prefix (LCP) between  $CT_k[i]$  and  $CT_k[j]$ , where  $i, j \in [1, N]$ . Further, let  $Height$  be a size- $N$  vector, where  $Height[i]$  denotes the height of  $L_k[i]$  that equals to the length of the LCP between the two  $k$ -order contexts of  $CT_k[i-1]$  and  $CT_k[i]$ , i.e.  $Height[i] = \|lcp(i-1, i)\|$  ( $\|\cdot\|$  is the cardinality operator for a set, returning the set's size.).

**Lemma 1.** *Given  $M_k$ , the following items regarding  $D$  and  $Height$  are equivalent.*

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

From the above lemma, it is easy to see that once the vector  $Height$  is available,  $D$  can be easily computed in  $O(N)$  by traversing  $Height$  once. Intuitively, this implies that we can convert the problem of solving  $D$  to finding an efficient solution for the computation of  $Height$ .

### Cycle of Characters

Now, we introduce the definition of *cycle*, which builds the foundation for developing our algorithm to compute the vector  $Height$  in linear time/space.

**Definition 4.** *Cycle  $\alpha(i)$ : the list of characters consisting of a subset of the characters in  $L_k$ , satisfying*

$$\alpha(i) = \begin{cases} \alpha(i)[1] = L_k[i] = L_k[Q_k[j]], & \text{for } \alpha(i)[\|\alpha(i)\|] = L_k[j]; \\ \alpha(i)[x+1] = L_k[Q_k[j]], & \text{for } \alpha(i)[x] = L_k[j] \text{ and } x \in [1, \|\alpha(i)\| - 1]; \end{cases}$$

Given  $\alpha(i)$ , calling the function  $IBWT(L_k, i, \|\alpha(i)\|)$  will backward retrieve all the characters in  $\alpha(i)$  one by one with a period length of  $\|\alpha(i)\|$ . In this sense, we term  $\alpha(i)$  a cycle. From the definition of cycle, we immediately see this property.

*Property 4.* Any two cycles are disjoint.

### Finding All Cycles

Because of the existence of cycles, we want to discover all the cycles from  $L_k$  first and then we use them to compute the heights for all characters in  $L_k$  in a linear complexity of  $O(N)$ . To achieve this goal, we introduce three one-dimension size- $N$  arrays  $X_0$ ,  $X_1$  and  $Y$  to store all the cycles. An algorithm for finding all the cycles from  $L_k$  in  $O(N)$  time/space is given below.

1. Initially, mark all items of  $L_k$  as unvisited.
2. Traverse  $L_k$  once from left to right. 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  $X_0$ . To help separate two neighbor cycles stored in  $X_0$ , we maintain the relative head and tail positions of each cycle by  $X_1$ . The array  $X_1$  contains two different kinds of values: non-negative values and negative values. If  $X_1[i] \geq 0$ ,  $X_0[i + X_1[i]]$  is the end character of the cycle; otherwise,  $X_0[i + X_1[i]]$  is the head character of the cycle. To show where the characters in  $X_0$  comes from  $L_k$ , the array  $Y$  is used to map each character in  $L_k$  to its position in  $X_0$ , i.e.  $L_k[Q_k[i]] = X_0[Y[i]]$ . Doing in this way, extracting all the cycles from  $L_k$  and compute the arrays  $X_0$ ,  $X_1$  and  $Y$  can be done in a total time/space complexity of  $O(N)$ .

Let the immediate predecessor of  $S_i$  in  $M_0$  be  $S_{i-1}$  for  $i \in [2, N]$ , and  $S_N$  for  $i = 1$ , respectively. To generalize these definitions, in  $M_0$ , we call the  $j$ th row the  $x$ th generation successor of the  $i$ th row and the  $i$ th row the  $x$ th generation predecessor of the  $j$ th row, if the  $j$ th row is  $x$  rows cyclically below the  $i$ th row. Let  $Q_k^x$  be the power notation of  $Q_k$ , which maps each row in  $M_k$  to its  $x$ th generation successor (recalling that  $Q_k$  is the vector mapping each row in  $M_k$  to its immediate successor), for instance,  $Q_k^3[i] = Q_k[Q_k[Q_k[i]]]$ . Similarly, we can define  $P_k^x[i]$  to map row  $i$  in  $M_k$  to its  $x$ th generation predecessor. From the definitions of  $CT_k$ ,  $L_k$ ,  $Q_k$ ,  $P_k$  and cycle, we observed these properties.

*Property 5.* For any  $k \in [1, N]$ ,  $i \in [1, N]$ , we have (1)  $Q_k^{\|\alpha(i)\|}[i] = P_k^{\|\alpha(i)\|}[i] = i$ ; and (2)  $CT_k[i] = [L_k[Q_k[i]], L_k[Q_k[Q_k[i]]], \dots, L_k[Q_k^k[i]]]$ ;

*Property 6.* The  $k$ -order context of  $L_k[i]$  is the first  $k$  characters of the string made up of the unlimited repetitions of cycle  $\alpha(Q_k[i])$ .

Property 6 describes a relationship between the context of a character in  $L_k$  and the cycle that the character belongs to. According to the definition of cycle, we have  $\alpha(Q_k[i]) = [L_k[Q_k[i]], L_k[Q_k[Q_k[i]]], \dots, L_k[Q_k^{\|\alpha(Q_k[i])\|}[i]]$ ; from Property 5, we have  $CT_k[i] = [L_k[Q_k[i]], L_k[Q_k[Q_k[i]]], \dots, L_k[Q_k^k[i]]]$ . Comparing  $CT_k[i]$  and  $\alpha(i)$ , this property is immediately true because the cycle  $\alpha(Q_k[i])$  will repeat itself in  $CT_k$  at each position  $j \in [1, k]$  satisfying  $(j-1)\% \|\alpha(Q_k[i])\| = 0$ , where  $\%$  is the integer modulo operator. Once we have all the cycles extracted from  $L_k$  and have them saved in the three arrays  $X_0$ ,  $X_1$  and  $Y$ , retrieving the character that is  $l$  character(s) cyclic far away from the character  $X_0[i]$  is a simple algebra problem, which can be trivially done in time  $O(1)$ . In other words, for any given character  $L_k[i]$ , using  $X_0$ ,  $X_1$  and  $Y$ , we can retrieve the character that is  $l$  character(s) cyclic away from it in the original text  $S$  in time  $O(1)$ , where  $l \in [0, k-1]$ .

## Computing Heights for a Cycle

We establish a theorem for inductive computing the heights of all the characters in a cycle, as below.

**Theorem 1.** *Height[ $Q_k[i]$ ]  $\geq$  Height[ $i$ ] - 1 for any  $k \in [1, N]$ ,  $i \in [2, N]$  and Height[ $i$ ]  $\geq 1$ .*

*Proof.* Given that Height[ $i$ ] =  $\|lcp(i-1, i)\| \geq 1$ , according to the definition of  $Q_k$ , we have  $Q_k[i-1] < Q_k[i]$  and it is trivial to see that Height[ $j$ ]  $\geq$  Height[ $i$ ] - 1 for any  $j \in [Q_k[i-1], Q_k[i]]$ .

Given the notations in this paper and Theorem 1, we can derive from [12] the linear algorithm GetHeight<sup>5</sup> in Fig. 3 for computing the LCPs for the characters

<sup>5</sup> Please notice that GetHeight can not be used to compute the LCPs for any two characters belonging to two different cycles! How to compute the LCPs for any two characters in two different cycles is the critical part of our solution.

in a *single cycle*, where  $Pos$  is a size- $N$  vector with each  $Pos[i]$  giving the position index of  $L_k[i]$  in the original text  $S$ , i.e.  $S[Pos[i]] = L_k[i]$ . However, neither  $S$  nor  $Pos$  is known, for how to retrieve  $S$  is the ultimate goal of the problem in our hands. To solve this problem, we can utilize  $X_0$ ,  $X_1$  and  $Y$  to do the same job instead. Notice that in `GetHeight`, there is  $h < k$ , which implies that both  $S[Pos[j] + h]$  and  $S[Pos[j - 1] + h]$  in line 6, as well as  $Height[Pos[j]]$  in line 10, always can be retrieved/maintained in  $O(1)$  time using  $X_0$ ,  $X_1$  and  $Y$ . In other words, we can derive another logically equivalent alternative for `GetHeight` that can do the same job without knowing  $S$  and  $Pos$ . (Details are omitted here due to space limit.)

The complexity of `GetHeight` is dominated by the execution time of line 8 in the inner loop. If we comment out lines 11-12, it is obvious that `GetHeight` has a time complexity of  $O(k)$ , for  $h$  can be increased one at a time for at most  $k$  times by line 8. By adding lines 11-12 back to `GetHeight`, it can trigger at most  $n$  more times running of line 8. This is because line 12 can be executed at most  $n$  time, and each running of line 12 can enable one more running of line 8, therefore at most  $n$  times running of line 8 can be introduced to the running time complexity. As the combined consequence of line 8 and lines 11-12, the total time complexity of `GetHeight` is  $O(k + n)$ , which is  $k$ -dependent.

```

GetHeight(int start, int n) {
1  j=start; h=0;
2  for(i=1; i<=n; i++)
3  {
4    while(h<k) // k-order LCP is at most k.
5    {
6      if(j==1 || S[Pos[j]+h+1] != S[Pos[j-1]+h+1])
7        break;
8      h++;
9    }
10   Height[Pos[j]]=h; // save the LCP for the char S[Pos[j]].
11   if(h>0)
12     h--; // decrease for computing the LCP of the succeeding char.
13   j=Qk[j]; // update the index of the succeeding char.
14 }
15 return Height;
}

```

**Fig. 3.** Algorithm for computing the heights for all the characters in a single cycle  $\alpha(start)$

**Theorem 2.** For any cycle  $\alpha(i)$ ,  $i \in [1, N]$ , the heights of all characters in  $\alpha(i)$  can be computed in a time complexity of  $O(k + \|\alpha(i)\|)$ .

*Proof.* The correctness of this theorem comes directly from the above analysis for the time complexity of GetHeight, for  $n = \|\alpha(i)\|$  in this case.

From the above result, we know that for each cycle  $\beta$  stored in  $X_0$ , we can compute the heights of all characters in  $\beta$  in a time complexity of  $O(k + \|\beta\|)$ . Hence, the total time complexity of computing the heights for all the characters in all cycles is  $\sum_{\beta_i} (k + n_i)$ , where  $n_i = \|\beta_i\|$ , which can be decomposed into two parts as below: one for the cycles not longer than  $ck$  and another for those longer than  $ck$ , where  $c > 0$ ,

$$O\left(\sum_{n_i \leq ck} (k + n_i)\right) + O\left(\sum_{n_j > ck} (k + n_j)\right).$$

For a cycle longer than  $ck$ ,  $k$  in the complexity can be safely ignored from  $O(\sum_{n_j > ck} (k + n_j))$ , resulting in  $O(\sum_{n_j > ck} (k + n_j)) = O(\sum_{n_j > ck} n_j) = O(N_{ck})$ , where  $N_{ck}$  is the total number of characters in all cycles longer than  $ck$  (cf. Property 4). However, for a cycle not longer than  $ck$ ,  $k$  in the complexity can not be ignored, because the complexity is  $O(\sum_{n_i \leq ck} (k + n_i)) = O(\sum_{n_i \leq ck} k)$ , which is a function of  $k$ . To exclude  $k$  from the complexity, we need to explore more properties of cycles.

**Lemma 2.** *For a cycle  $\beta$  longer than  $ck$ , where  $c > 0$  is a constant, the heights of all its characters can be computed in a time complexity of  $O(\|\beta\|)$ .*

*Proof.* According to Theorem 2, we can compute the heights for all characters in the cycle  $\beta$  with a time complexity of  $O(k + \|\beta\|)$ . Given that  $\|\beta\| > ck$ , we have  $O(k + \|\beta\|) = O(\|\beta\|)$ .

### Computing Heights for All Cycles

Let  $CH_0$  denote the subset of the characters in all the cycles longer than  $ck$ , where  $c > 0$  is a constant. In addition, let  $CH_1$  denote the subset of the characters in  $L_k$ , satisfying that for each  $L_k[i] \in CH_1$ ,  $L_k[i-1]$  is in  $CH_0$ , where  $i \in [2, N]$ . Following the definitions of  $CH_0$  and  $CH_1$ , it is trivial to derive from Lemma 2 the below lemma about the complexity of computing the heights of all characters in  $CH_0 \cup CH_1$ .

**Lemma 3.** *The heights of all characters in  $CH_0 \cup CH_1$  can be computed in a time complexity of  $O(\|CH_0\| + \|CH_1\|)$ .*

*Proof.* Referring to the algorithm GetHeight in Fig.3, at line 6, the character  $S[Pos[j] + h + 1]$  is compared with  $S[Pos[j - 1] + h + 1]$  to compute the height of  $L_k[j]$ . Similarly, if we revise the code at Line 6 to be “`if(j==N || S[Pos[j]+h+1]!=S[Pos[j+1]+h+1])`”, i.e. to compare  $S[Pos[j] + h + 1]$  with  $S[Pos[j + 1] + h + 1]$  instead of  $S[Pos[j - 1] + h + 1]$ , and revise Line 10 to be “`Height[Pos[j+1]]=h`”, the algorithm GetHeight can compute the heights of all the characters in  $CH_1$ . Hence, according to Lemma 2, the heights of all characters in set  $CH_0 \cup CH_1$  can be computed in a time complexity of  $O(\|CH_0\| + \|CH_1\|) = O(\|CH_0\|)$ .

Having solved the issue of computing the heights of all the characters in the set  $CH_0 \cup CH_1$  in a linear time complexity, the only pending problem is how to compute the heights of all the other remaining characters in the set  $\{L_k[i] | i \in [1, N]\} - \{CH_0 \cup CH_1\}$  in a linear time complexity independent of  $k$ . For this purpose, we establish the below theorems and lemmas.

**Theorem 3.** *For any  $i \in [2, N]$ ,  $CT_k[i] = CT_k[i-1]$ ,  $\|\alpha(i)\| \leq \lfloor k/2 \rfloor$  and  $\|\alpha(i-1)\| \leq \lfloor k/2 \rfloor$ , we have  $\alpha(i) = \alpha(i-1)$ .*

*Proof.* Without loss of generality, let's suppose  $\|\alpha(i-1)\| \geq \|\alpha(i)\|$ . We continue the proof as follows:

1. Given the condition for the theorem in question, we have that  $M_k[i, 1 : \|\alpha(i)\|] = M_k[i-1, 1 : \|\alpha(i)\|]$  and  $M_k[i, \|\alpha(i-1)\| + 1 : \|\alpha(i-1)\| + \|\alpha(i)\|] = M_k[i-1, \|\alpha(i-1)\| + 1 : \|\alpha(i-1)\| + \|\alpha(i)\|]$ . Because that  $M_k[i-1, 1 : \|\alpha(i)\|] = M_k[i-1, \|\alpha(i-1)\| + 1 : \|\alpha(i-1)\| + \|\alpha(i)\|]$ , we have  $M_k[i, 1 : \|\alpha(i)\|] = M_k[i, \|\alpha(i-1)\| + 1 : \|\alpha(i-1)\| + \|\alpha(i)\|]$ . Hence, the preceding character of  $M_k[i, 1]$  is  $M_k[i, \|\alpha(i-1)\|]$ , i.e.  $L_k[i] = M_k[i, \|\alpha(i-1)\|] = M_k[i-1, \|\alpha(i-1)\|] = L_k[i-1]$ . Given  $L_k[i] = L_k[i-1]$  and  $CT_k[i] = CT_k[i-1]$ , we further have  $CT_k[P_k[i]] = CT_k[P_k[i-1]]$ , i.e. the two  $k$ -order contexts at rows  $P_k[i]$  and  $P_k[i-1]$  in  $M_k$  are equivalent.
2. From the above analysis (in this proof) and Property 5, we see  $L_k[P_k^{\|\alpha(i)\|}[i]] = L_k[i]$  as well as  $L_k[P_k^{\|\alpha(i)\|}[i-1]] = L_k[i-1]$ . Furthermore, for any character  $L_k[j] \in \alpha(i)$ , we have  $L_k[j-1] \in \alpha(i-1)$ ,  $L_k[j] = L_k[j-1]$  and  $CT_k[j] = CT_k[j-1]$ . Hence,  $\alpha(i)$  and  $\alpha(i-1)$  are two equivalent cycles.

The above theorem says that for any two characters  $i$  and  $i-1$  in  $L_k$ , if the  $k$ -order contexts of the two characters are equal and both cycles  $\alpha(i)$  and  $\alpha(i-1)$  are not longer than  $\lfloor k/2 \rfloor$ , then the two cycles must be equivalent.

**Lemma 4.** *For any  $i \in [2, N]$ , if  $\|\alpha(L_k[i-1])\| \leq \lfloor k/2 \rfloor$  and  $\|\alpha(L_k[i])\| \leq \lfloor k/2 \rfloor$ , then  $CT_k[i-1] = CT_k[i]$  only if the lengths of the two cycles  $\alpha(i-1)$  and  $\alpha(i)$  are equal.*

*Proof.* Given the condition, in case that  $CT_k[i-1] = CT_k[i]$ , from Theorem 3, we have  $\alpha(i-1) = \alpha(i)$ , which implies that  $\|\alpha(i-1)\| = \|\alpha(i)\|$ .

Lemma 4 suggests that if two cycles  $\alpha(i)$  and  $\alpha(i-1)$  are not longer than  $\lfloor k/2 \rfloor$  and their lengths are different, we can immediately determine that the two  $k$ -order contexts of  $L_k[i]$  and  $L_k[i-1]$  are different in a complexity of  $O(1)$ , as stated below.

**Corollary 1.** *For any two cycles  $\alpha(i-1)$  and  $\alpha(i)$  not longer than  $\lfloor k/2 \rfloor$ ,  $i \in [2, N]$ , we have  $CT_k[i-1] \neq CT_k[i]$  if the lengths of two cycles are different.*

*Proof.* According to Lemma 4, provided that the two cycles are not longer than  $\lfloor k/2 \rfloor$ , the two  $k$ -order contexts of  $L_k[i]$  and  $L_k[i-1]$  can be identical only if the lengths of the two cycles are equal. Hence, if the two cycles  $\alpha(i-1)$  and  $\alpha(i)$  have different lengths, we must have  $CT_k[i-1] \neq CT_k[i]$ .

The above corollary says that if the lengths of two cycles  $\alpha(i)$  and  $\alpha(i-1)$  are not longer than  $\lfloor k/2 \rfloor$  and different, then the  $k$ -order contexts of  $L_k[i-1]$  and  $L_k[i]$  are different too.

**Corollary 2.** *For any  $i \in [2, N]$ ,  $CT_k[i] = CT_k[i-1]$ ,  $\|\alpha(i)\| \leq \lfloor k/2 \rfloor$  and  $\|\alpha(i-1)\| \leq \lfloor k/2 \rfloor$ , we have  $CT_k[j] = CT_k[j-1]$  and  $\alpha(j) = \alpha(j-1)$  for any character  $L_k[j] \in \alpha(i)$ .*

*Proof.* Given  $CT_k[i] = CT_k[i-1]$ ,  $\|\alpha(i)\| \leq \lfloor k/2 \rfloor$  and  $\|\alpha(i-1)\| \leq \lfloor k/2 \rfloor$ , from the proof of Theorem 3, we know that  $L_k[i] = L_k[i-1]$  and  $\alpha(i) = \alpha(i-1)$ . Further, according to Property 6, we have  $CT_k[P_k[i]] = CT_k[P_k[i-1]]$ . Because that when  $k \in [1, N]$  and  $L_k[i] = L_k[i-1]$ , there must be  $P_k[i-1] = P_k[i] - 1$ . Hence, from Theorem 3 again, we have that  $CT_k[P_k[i]] = CT_k[P_k[i]-1]$  and both cycles  $\alpha(P_k[i])$  and  $\alpha(P_k[i]-1)$  are equal and not longer than  $\lfloor k/2 \rfloor$ . Repeating the induction in the same way, we have that for any character  $L_k[j] \in \alpha(i)$ , we have  $CT_k[j] = CT_k[j-1]$  and  $\alpha(j) = \alpha(j-1)$ .

The above corollary says that for any  $L_k[i]$  belonging to a cycle not longer than  $\lfloor k/2 \rfloor$ , if the cycle  $\alpha(i-1)$  is also not longer than  $\lfloor k/2 \rfloor$ , and the two  $k$ -order contexts of  $L_k[i]$  and  $L_k[i-1]$  are equivalent, then for any character  $L_k[j]$  belonging to the cycle  $\alpha(i)$ , the  $k$ -order context of  $L_k[j]$  must equal to that of  $L_k[j-1]$ .

**Corollary 3.** *For  $i \in [2, N]$ ,  $CT_k[i] \neq CT_k[i-1]$ ,  $\|\alpha(i)\| \leq \lfloor k/2 \rfloor$  and  $\|\alpha(i-1)\| \leq \lfloor k/2 \rfloor$ , we have  $CT_k[j] \neq CT_k[j-1]$  for any character  $L_k[j] \in \alpha(i)$  seeing  $\|\alpha(j-1)\| \leq \lfloor k/2 \rfloor$ .*

*Proof.* We prove it by contradiction. Suppose that there exists a character  $L_k[j] \in \alpha(i)$  seeing  $CT_k[j] = CT_k[j-1]$  and  $\|\alpha(j-1)\| \leq \lfloor k/2 \rfloor$ . Because  $\|\alpha(j)\| = \|\alpha(i)\| \leq \lfloor k/2 \rfloor$ , from Corollary 2, we have that for any  $L_k[x] \in \alpha(j)$ , there must be  $CT_k[x] = CT_k[x-1]$ . This contradicts to the assumption  $CT_k[i] \neq CT_k[i-1]$  of this corollary, where  $L_k[i] \in \alpha(j)$ .

The above corollary says that for any  $L_k[i]$  in a cycle not longer than  $\lfloor k/2 \rfloor$ , if the cycle  $\alpha(i-1)$  is also not longer than  $\lfloor k/2 \rfloor$  and the  $k$ -order contexts of  $L_k[i]$  and  $L_k[i-1]$  are different, then for any character  $L_k[j] \in \alpha(i)$  and  $\alpha(j-1)$  is not longer than  $\lfloor k/2 \rfloor$ , the two  $k$ -order contexts of  $L_k[j]$  and  $L_k[j-1]$  must be different too.

**Theorem 4.** *For any cycle  $\beta$  not longer than  $\lfloor k/2 \rfloor$ , the values in  $D$  for all characters in set  $\eta = \{L[j] \in \beta | j \in [2, N] \text{ and } \|\alpha(j-1)\| \leq \lfloor k/2 \rfloor\}$  can be computed in a complexity of  $O(\|\beta\|)$ .*

*Proof.* For each character  $L_k[j]$  in  $\beta$ , there are two cases with respect to the length of cycle  $\alpha(j-1)$ : longer than  $\lfloor k/2 \rfloor$  or not. The former case is for the characters in  $CH_1$  (cf. Lemma 3), thus has been considered there. As for the later case, according to Theorem 3 and Corollary 1, we need at most  $O(\|\beta\|)$ , instead of  $O(k + \|\beta\|)$ , steps to compare all the characters in the two contexts of  $L_k[j]$

and  $L_k[j - 1]$  to determine if they are equal or not when  $\|\alpha(j)\| = \|\alpha(j - 1)\|$ . Based on the comparison result, from Corollary 2 and 3, we can extend the result to all the other characters in  $\eta$  with at most  $\|\beta\| - 1$  steps, where each step has a complexity of  $O(1)$ . Hence, we complete the proof.

### Making the Solution

Now, we apply the analysis results established in the previous subsections to build a solution for computing  $D$  in linear time/space. The key idea is to decompose the problem of computing  $D[i]$  for each  $L_k[i]$ ,  $i \in [1, N]$ , into two sub-problems, depending on whether  $\alpha(i)$  is longer than  $\lfloor ck \rfloor$ . In particular, we choose the constant  $c = 1/2$  in our algorithm design. The algorithm consists of the following 3 steps:

1. Compute  $P_k$  and  $Q_k$  from  $L_k$ , which can be done in  $O(N)$  time.
2. Find all cycles from  $L_k$  using  $P_k$  and  $Q_k$ , and record by  $X_0$ ,  $X_1$  and  $Y$ , which can be done in  $O(N)$  time.
3. Now, let's initialize all the items of  $L_k$  as unvisited and then traverse  $L_k$  once. For each unvisited  $L_k[i]$ , we mark all the characters in  $\alpha(i)$  as visited in  $O(\alpha(i))$  time. Further, according to whether  $\alpha(i)$  is longer than  $ck$  (which can be determined in time  $O(1)$  using  $X_0$ ,  $X_1$  and  $Y$ ) or not, we do according to the following two cases:
  - The cycle  $\alpha(i)$  is longer than  $ck$ . We calculate  $D[i]$  for any character  $L_k[i]$  in the cycle together with  $D[i + 1]$  for  $L_k[i + 1]$ . The total time for this step is well bounded by the number of characters in all the cycles longer than  $ck$  (cf. Lemma 3), thus bounded by  $O(N)$ .
  - For any  $L_k[j]$  with  $\alpha(j)$  not longer than  $ck$ , depending on whether the cycle  $\alpha(j - 1)$  is longer than  $ck$  or not, we further consider two subcases.
    - (1) True. This has been considered in the previous case for the characters in cycles longer than  $ck$ .
    - (2) False. We will look into the lengths of both cycles  $\alpha(j)$  and  $\alpha(j - 1)$ .
      - If the two cycles' lengths are different from each other, we can determine that the two contexts  $CT[j]$  and  $CT[j - 1]$  must not equal to each other (cf. Corollary 1), in a time complexity  $O(1)$ . To propagate the same result to all the other characters in the cycle whose corresponding values in vector  $D$  have not been computed (cf. Corollary 3), the time complexity is  $O(\|\alpha(j)\|)$ .
      - If both cycles  $\alpha(j)$  and  $\alpha(j - 1)$  have the same length, to compute  $D[j]$ , we only need to consider a length up to the size of the cycle  $\alpha(j)$  (cf. Property 6), instead of the context order  $k$ . Once  $D[j]$  has been computed, the same value of  $D[j]$  can be populated to the other characters in cycle  $\alpha(j)$  whose corresponding values in vector  $D$  have not been computed (cf. Corollary 2 and 3), in a time complexity  $O(\|\alpha(j)\|)$ .

Because all cycles are disjoint, the aggregated number of all characters in all such kinds of cycles is bounded by  $O(N)$ , the time complexity for this case is thus bounded by  $O(N)$ .

Hence, the total time complexity counted for this case is  $O(N)$ .

The algorithm described above can always compute  $D$  within  $O(N)$  time complexity, no matter what kind of combination of cycles we have. The space complexity is obvious  $O(N)$ , for only a constant number of size- $N$  arrays are required. As a result of the above analysis, we have the following theorem to state the linearity of our algorithm for computing  $D$  as well as the inverse ST for any given  $L_k$ .

**Theorem 5.** *Given  $L_k$ , we can restore the original text of  $S$  in  $O(N)$  time/space for any  $k \in [1, N]$ .*

The presented algorithm has been coded in C and validated, which is available upon request.

## Acknowledgment

The authors wish to thank the anonymous reviewers for this paper and its under-review journal version, for their constructive suggestions and insightful comments that have helped improve the presentation of this paper.

## References

1. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124, Digital Systems Research Center, California USA (May 1994)
2. Schindler, M.: The sort transformation. [Online] Available: <http://www.compressconsult.com>
3. Schindler, M.: A fast block-sorting algorithm for lossless data compression. In: Proceedings of DCC'97. (1997) 469
4. Schindler, M.: Method and apparatus for sorting data blocks. Patent in United States (6199064) (March 2001)
5. Nong, G., Zhang, S.: Unifying the Burrows-Wheeler and the Schindler transforms. In: Proceedings of DCC'06. (March 2006) 464
6. Nong, G., Zhang, S.: An efficient algorithm for the inverse ST problem. In: Proceedings of DCC'07. (2007) 397
7. Nong, G., Zhang, S.: Efficient algorithms for the inverse sort transform. IEEE Transactions on Computers **56**(11) (November 2007) 1564–74
8. Yokoo, H.: Notes on block-sorting data compression. Electronics and Communications in Japan (Part III: Fundamental Electronic Science) **82**(6) (1999) 18–25
9. Bird, R.S., Mu, S.C.: Inverting the burrows-wheeler transform. Journal of Functional Programming **14**(6) (November 2004) 603–612
10. Manzini, G.: The Burrows-Wheeler transform: theory and practice. In: LNCS. Volume 1672. (September 1999) 34–47
11. Balkenhol, B., Kurtz, S.: Universal data compression based on the Burrows-Wheeler transformation: theory and practice. IEEE Transactions on Computers **49**(10) (October 2000) 1043–53
12. Kasai, T., Lee, G., Arimura, H., et al.: Linear-time longest-common-prefix computation in suffix arrays and its applications. LNCS **2089/2001** (July 2001) 181–192