Efficient Algorithms for the Inverse Sort Transform

Ge Nong, *Member*, *IEEE*, and Sen Zhang, *Member*, *IEEE*

Abstract—As an important variant of the Burrows-Wheeler Transform (BWT), the Sort Transform (ST) can speed up the transformation by sorting only a portion of the matrix. However, because the currently known inverse ST algorithms need to retrieve the complete k-order contexts and use hash tables, they are less efficient than the inverse BWT. In this paper, we propose three fast and memory-efficient inverse ST algorithms. The first algorithm uses two auxiliary vectors to replace the hash tables. The algorithm achieves $O(kN)$ time and space complexities for a text of N characters under the context order k. The second uses two additional compact "alternate vectors" to further eliminate the need to restore all of the k-order contexts and achieve $O(N)$ space complexity. Moreover, the third uses a "doubling technique" to further reduce the time complexity to $O(N \log_2 k)$. The hallmark of these three algorithms is that they can invert the ST in a manner similar to inverting BWT in that they all make use of precalculated auxiliary mapping vectors and require no hash tables. These unifying algorithms can also better explain the connection between the BWT and the ST: Not only can their forward components be performed by the same algorithm framework, but their respective inverse components can also be efficiently conducted by the unifying algorithm framework proposed in the present work.

Index Terms—Burrows-Wheeler transform, inverse sort transform, limit-order contexts, algorithm design, data compression.

1 Introduction

The Burrows and Wheeler Transform (BWT) [1] can reorder a text into a more "compressible" sequence. Briefly, given a text S of N characters, the BWT proceeds in three steps: 1) to derive a matrix consisting of N rotations (cyclic shifts) of S, 2) to sort the rows of the matrix lexicographically, and 3) to extract the last column of the sorted matrix to produce the transformed text. The BWT itself does not reduce the length of a given text; however, the transformed text tends to group identical characters together so that the probability of finding them clustered together is substantially increased. As a consequence, the transformed text could be better compressed by fast locally adaptive encoding algorithms such as run-length encoding and move-to-front coding in combination with Huffman or arithmetic coders.

The BWT and its variants have found many applications in a broad range of fields: DNA sequence compression [2], compression of electrocardiogram (ECG) signals [3], compression algorithms in communication systems [4], and image compression [5], [6]. The success of the BWT and its variants has also stimulated research on exploring their underlying information and mathematical theories. Balkenhol et al. provided a thorough analysis of the BWT from an information theory perspective [7], [8]. Manzini investigated the relationship between the compression ratio and the empirical entropy of the input text and systematically analyzed why the BWT can benefit text compression in general [9], [10]. Arnavut and Arnavut investigated the multiset permutation property of the BWT from the mathematical explanation point of view and proposed a new transformation called Linear Ordering Transformation [11], [12]. Yokoo explored the combinatorial properties of the BWT from the perspective of reversibility of transformation [13].

One major drawback of the BWT is that any naive implementation of the second step can result in a superlinear sorting time complexity that constitutes an evident bottleneck for the BWT. To remedy the drawback, domain scientists have used some clever sorting algorithms that exploit advanced data structures such as suffix tree, suffix array, and their variants [1], [7], [14], [15], [16], [17], [18] to speed up the transformation. For example, Larsson [19] has shown that an efficient $O(N \log^2 N)$ suffix-array-based algorithm [14], originally proposed by Sadakane, can achieve a worst-case time complexity of $O(N \log N)$. Hongo and Yokoo [21] used the Karp-Miller-Rosenberg (KMR) algorithm [20] to further reduce the asymptotic time complexity of the forward BWT to $O(N \log_3^2 N)$, although this is practically slower than Sadakane's algorithm. Recently, Puglisi et al. carried out a thorough performance study on the algorithms built on different suffix data structures [22], which confirmed that, in general, these suffix-array-based methods [14] not only outperform the linear algorithms [16], [18] but also are much faster than the suffix-tree-based methods.

Another approach for speeding up the transformation is through reducing the problem size by sorting only a portion of the matrix. This solution was due to Schindler and has been known as the Sort Transform (ST) [8], [9], [13], [15], [23], [24]. The basic idea of the ST is built upon the concept

- *G. Nong is with the Computer Science Department, Sun Yat-Sen University, Guangzhou, 510275, People's Republic of China. E-mail: issng@mail.sysu.edu.cn,*
- *S. Zhang is with the Department of Mathematics, Computer Science, and Statistics, State University of New York, College at Oneonta, Oneonta, NY 13820. E-mail: zhangs@oneonta.edu.*

Manuscript received 29 Aug. 2005; revised 22 May 2006; accepted 24 Apr. 2007; published online 22 June 2007.
Recommended for acceptance by F. Dehne.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0287-0805.
Digital Object Identifier no. 10.1109/TC.2007.70762.

of limited order and unlimited order contexts, where the full context of a character in a text is its suffix. The context of a character can have different limited orders ranging from 0 to $N-1$ and the full text is deemed the unlimited order context. Hence, the ST resembles the BWT but with the sorting part of the matrix limited to the k-order ($k \in [0, N]$) contexts only. The reduced comparison length accelerates the transformation. Schindler has used the ST to build a compression software *szip* [25]. In addition, it is also worth noting that, due to its deterministic time/space complexity, $O(kN)$, the ST is more suitable for memory-critical hardware or VLSI implementation [15], [23], especially when the context order k is small.[1] For example, consider an 802.11x wireless local area network operating in the infrastructure mode, where power saving in the mobile terminals is a major concern, while the base station has an unlimited power supply and far more powerful computing resources. In this case, we can use the ST for a mobile terminal to compress data to be uploaded to the base station and, meanwhile, use the BWT for the base station to compress data to be delivered to a mobile terminal. As a result, without degrading the compression efficiency, the mobile terminal can take advantage of both the fast ST and the fast inverse BWT. Another promising application for the ST is a new data encryption scheme built upon the ST with the transform order serving as the key, on which our intensive study is undergoing.

A major trade-off for the ST to achieve the speedup gain over the BWT is that the inverse ST is more complicated 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 can no longer be guaranteed. To deal with this situation, Schindler proposed a hash-based approach where the text retrieval has to rely on a hash-table-based context lookup, which in turn has to rely on the complete retrieval of all of the k-order contexts. As a contrast, neither of them is required by the inverse BWT. Inspired by this observation, we have proposed the idea of using an auxiliary vector-based framework [28], [29] to solve the inverse ST problem, which is different from any possible hashing-based approaches suggested by Schindler [23], [24] or Yokoo [13] but is similar to that used for the inverse BWT [1]. Since this framework does not use hashing tables, it requires only $O(N)$ space. However, the runtime complexity remains $O(kN)$. As continuing research of this auxiliary vector-based framework, we recently developed a new algorithm that can achieve time complexity $O(N \log_2 k)$ and space complexity $O(N)$. The details of both the $O(kN)$ and the $O(N \log_2 k)$ algorithms are presented here.

The remainder of this paper is organized as follows: In Section 2, we introduce some assumptions and notations that will be used in the subsequent sections. In Section 3, we revisit both the BWT and the ST more formally and present some mathematical explanations for them. In Section 4, we first present several properties and theorems and then, based on them, propose three fast algorithms to invert ST in a way similar to that of inverting BWT. Finally, we conclude the paper in Section 5.

1. The experiments in [14] showed that, for the Calgary [26] and the Canterbury [27] corpora, the average match lengths of two suffixes/contexts range from 3 to 25 characters, which implies that, in practice, a moderate value of k such as 16 to 32 is sufficient for the ST to approximate the compression boosting efficiency of the BWT.

2 Notations

A set of notations and simplifying assumptions used in [7] and other previous publications in this field is adopted. Unless otherwise specified, the forward BWT is simply referred to as the BWT and the backward BWT as the inverse BWT. The same notation convention applies to the ST and the inverse ST. For any integer numbers $l, r \in \mathbb{N}_0$, $[l, r]$ denotes the set $\{i \in \mathbb{N}_0 : l \le i \le r\}$, that is, a consecutive range or an interval of integers. The notation $Z[N_r, N_c]$ represents 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 2D subarray of $Z[N_r, N_c]$ covering the rows from a to b and the columns from c to d, where $1 \le a \le b \le N_r$ and $1 \le c \le d \le N_c$. In case $a = b$ and/or $c = d$, the simpler forms of $Z[a, c : d]$ or $Z[a : b, c]$ and $Z[a, c]$ can be used instead, respectively. Without loss of generality, a text of length N is denoted as $x_1 x_2 x_3 \ldots x_{N-1}\$$, where each character $x_i \in \Sigma, i \in [1, N-1]$ and Σ is the alphabet. The last character of the text is a *sentinel* \$, which is the lexicographically greatest character in the alphabet and which occurs exactly once in the text. Readers who are interested in the details of appending sentinels to text are referred to [7], [9], [30] since how to use sentinels is not the main subject of this paper.

Following the above simplifying assumptions, we now formally introduce a set of notations and definitions used throughout the paper.

- S. The row vector for a text $x_1 x_2 x_3 \ldots x_{N-1}\$$, denoted by $S = [\, x_1 \quad x_2 \quad x_3 \quad \cdots \quad x_{N-1} \quad \$\,]$. The size of the text is N. According to the cyclic rotation scheme, $S[i]$ is defined to be the immediate preceding character of $S[i+1]$, where $i \in [1, N-1]$, and $S[N]$ is the immediate preceding character of $S[1]$.
- OM. The $N \times N$ symmetric matrix originally constructed from the texts obtained by cyclically rotating the text S. Specifically, the first row of the matrix OM 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.
- $IP(S_i)$. The immediate predecessor of S_i, for any $i \in [1, N]$ with respect to the above OM. S_{i-1} is the immediate predecessor of S_i, where $i \in [2, N]$, and S_N is the immediate predecessor of S_1. Correspondingly, we call S_{i+1} the immediate successor of S_i, where $i \in [1, N-1]$, and S_1 is the immediate successor of S_N. To generalize the concepts, we define that the jth row is the xth generation successor of the ith row and the ith row is the xth generation predecessor of the jth row, if the jth row is x rows cyclically below the ith row in OM.
- k-order context. The k-order context of the last character of a text S is $S[1 : k]$, where $k \in [1, N]$. When $k = N$, a k-order context is also called the

unlimited order context; otherwise, it is called a limited order context.

- M_k. The sorted matrix of OM according to the k-order context used in the ST transform, where $k \in [1, N]$. When $k = N$, M_k can be simplified as M, representing the sorted matrix used in the BWT.
- S^{bwt}. The output of BWT on the input text S.
- $BWT(S)$ and $IBWT(S^{bwt})$. The BWT on S and the inverse BWT on S^{bwt}, respectively.
- S^{st-k}. The output of the k-order ST on the input text S.
- $ST(S, k)$ and $IST(S^{st-k})$. The k-order ST on S and the k-order inverse ST on S^{st-k}, respectively.
- F_k and L_k. The transposes of the first and the last columns of M_k, respectively, that is, $F_k = M_k[1 : N, 1]^T$ and $L_k = M_k[1 : N, N]^T$, where $k \in [1, N]$. When $k = N$, F_k and L_k will be simplified as F and L, respectively.
- I. The row index of the original text S in M_k.
- P_k. A row vector that maps the index of each character at L_k to its index at F_k, satisfying 1) $F_k[P_k[i]] = L_k[i]$, for $i \in [1, N]$, and 2) $P_k[i] < P_k[j]$, for $1 \le i < j \le N$, and $L_k[i] = L_k[j]$ or $L_k[i] < L_k[j]$, where $k \in [1, N]$. When $k = N$, the simpler form P is used. Although $P[i]$ guarantees finding the true immediate predecessor row for a given row i, $P_k[i]$ can only find a row sharing the same k-order context with the true immediate predecessor; thus, we say that $P_k[i]$ finds the pseudoimmediate predecessor for row i.
- Q_k. A row vector that maps the index of each character at F_k to its index at L_k, satisfying 1) $L_k[Q_k[i]] = F_k[i]$, for $i \in [1, N]$, 2) $Q_k[i] < Q_k[j]$ for $1 \le i < j \le N$ and $F_k[i] = F_k[j]$ or $F_k[i] < F_k[j]$, where $k \in [1, N]$. When $k = N$, the simpler form Q is used. Although $Q[i]$ guarantees finding the true immediate successor row for row i, $Q_k[i]$ can only find a row sharing the same k-order context with the true immediate successor; thus, we say $Q_k[i]$ finds the pseudoimmediate successor for row i.
- Q_k^x. The power notation of Q_k. Since Q_k is the vector that maps each row to its pseudoimmediate successor, Q_k^x is a vector that maps each row to its xth generation pseudosuccessor.
- C_k. A counter 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 ith row is new and is repeated for $C_k[i]$ times starting from the ith row consecutively to the $(i + C_k[i] - 1)$th row; otherwise, the k-order context at the ith row repeats the one at the previous row. A k-order context is new if either it is the first context (when $i = 1$) or it is different from that at the $(i-1)$th row (for $i \in [2, N]$). When a new k-order context is found, we say that there is a context switch. It is clear that the number of nonzero elements is the number of unique k-order contexts in M_k.
- T_k. An index 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 $L_k[i]$, starting from the $T_k[i]$th row, there are $C_k[T_k[i]]$ consecutive rows sharing the same k-order context with $L_k[i]$ being the first character.

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

Fig. 1. The matrix M resulting from the BWT on "mississippi$". From this matrix, we have $S^{bwt} = (L, I)$, where $L = ssmp\$pissiii$ and $I = 5$.

3 Revisiting the BWT and the ST

3.1 The Burrows-Wheeler Transform

As previously mentioned, the BWT is conducted in three steps. First, the $N \times N$ matrix OM is derived from the input S as follows: The first row is S and the remaining rows are constructed by applying successive cyclic left shifts to S. Second, all of the N rows of OM are sorted in lexicographical order. Without loss of generality, we assume that the sorting is in ascending order. The resulting matrix, denoted by M, is simply a permutation of OM with the reordered rows. Finally, the output of the BWT transform on S is denoted by $S^{bwt} = (L, I)$, where L is the transpose of the last column of M and I points to the row position of the original S in M. As an example, we show in Fig. 1 the result of the BWT on the text "mississippi$" (this sample input text will be used throughout the paper for illustration purpose).

The inverse BWT aims to restore the original text S from S^{bwt}. The solution for the inverse BWT [1] was developed based on the following properties:

Property 1. *For $j \in [1, N]$, $IP(S_i)[j]$ is the immediate preceding character of $S_i[j]$ in S.*

This property states that OM is a symmetric matrix.

Property 2. *Given L, F can be obtained by sorting all of the characters of L.*

Property 3. *In both F and L, the relative orders of any two identical characters are consistent.*

From Properties 2 and 3, it is easy to see that P can be computed by traversing L twice: One pass obtains F and the other populates P.

Property 4. *For $i \in [1, N]$, $M[P[i], 1 : N] = IP(M[i, 1 : N])$.*

Property 4 guarantees that, in the transformed matrix M, the row $M[P[i], 1 : N]$ is the immediate predecessor of the row $M[i, 1 : N]$. From the definition of P (see Section 2), we know that, for a given index i, $P[i]$ should be the index to trace back to, through which we can locate the character $L[P[i]]$, that is,

```
INVERSE-BWT(L, I)
    ▷ L is the last column vector
    ▷ I is the row index of S
1   len ← LENGTH(L)
2   j ← I
3   i ← len
4   while i > 0
5       do
6           S[i] ← L[j] ▷ restore the ith character.
            ▷ update the indexes j and i for retrieving the
            next preceding character.
7           j ← P[j]
8           i ← i − 1
9   return S
```

Fig. 2. The inverse BWT algorithm.

the immediate preceding character of $L[i]$, with respect to their positions in the original S (see Property 1).

Based on Property 4, the inverse algorithm [1] uses vector P to backtransform S^{bwt} to S. The pseudocode[2] of this classical inverse BWT algorithm is shown in Fig. 2, where the details for computing P are omitted due to its simplicity.

3.2 The ST

The ST adopts a two-hierarchy sorting scheme, that is, the lexicographical sorting criterion in tandem with the positional sorting criterion. Specifically, the k-order ST will sort all of the rows of OM according to their k-order contexts first. Then, if there are any two identical k-order contexts, the tie will be resolved by preserving the relative order between them in the original OM.

For example, if we apply the 4-order ST algorithm to "mississippi$", we will obtain M_4, as illustrated in Fig. 3, and $S^{st-4} = ([s\ \ m\ \ s\ \ p\ \ \$\ \ p\ \ i\ \ s\ \ s\ \ i\ \ i\ \ i], 5)$.

The inverse ST proposed by Schindler involves two phases: the k-order contexts retrieval and the original text retrieval. In phase one, all of the k-order contexts of M_k need to be restored. For $k = 1$, the contexts can be obtained by simply sorting the characters in L_k. Because L_k and S are two permuted instances of the same set of characters and L_k consists of all characters of S sorted by their k-order contexts, sorting the characters of L_k will result in all of the 1-order contexts being arranged in their correct orders. In case $k > 1$, restoring the contexts from L_k involves more steps, but it is essentially the same as what we do for $k = 1$. Let A and B be two arrays of N elements, where each element can accommodate a text up to k characters. Initially, each element of A and B is empty. The following two steps are repeated k times to restore all of the k-order contexts into A, where each iteration increases the context order by one:

1. $B[i] = L_k[i] \oplus A[i]$, for $i \in [1, N]$, where "$\oplus$" denotes the text concatenating operator.

2. We typeset the pseudocodes in the style of *Introduction to Algorithms*, second edition, by Cormen, Leiserson, Rivest, and Stein (CLRS).

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

Fig. 3. The matrix M_4 for the 4-order ST on "mississippi$".

2. Lexicographically sort all of the rows of array B into array A.

At Step 1, all elements of A have been sorted in lexicographical order. Hence, at Step 2, we only need to sort B according to its leftmost column, which can be done via the vector P. Given that this procedure will iterate k times, both the time and the space complexities of the algorithm are $O(kN)$.

In phase two, S will be derived from L_k based on the fully restored k-order contexts. Due to the ST's partial sorting strategy, Property 4 no longer holds for the ST. This is because, in BWT, when a character $L[i]$ is mapped to the character $F[P[i]]$, the mapping actually finds the whole immediate predecessor row of the ith row in M, but the same mapping in the ST can only find the k-order context of the immediate predecessor row for the ith row instead of its entire immediate predecessor row. Therefore, we cannot use $P[i]$ to correctly find out the true immediate predecessor row for a given row among multiple candidate rows that share exactly the same k-order context. To address this challenge, Schindler proposed a hashing-based strategy in [24], where he described the detailed hashing algorithms to invert ST for $k \leq 2$. For higher k, only a vague solution without details was mentioned.[3] The inverse ST algorithm summarized from [24] is described in Fig. 4 and the logic of the hash table strategy is supposed to be implemented by lines 13 to 15.

3.3 Remarks

The ST is generalized from the BWT by treating the BWT as a special case. When $k = N$, the N-order ST produces the same result as the BWT because both transforms sort all of the rows of the matrix OM to their full lengths. In this sense, the ST is an efficient alternative to the BWT and both can be performed using the same algorithm for the ST. It is also worth noting that this generalization shows the strong

3. In the first and the last paragraphs on the third page of the patent description section: "*(the first paragraph) In a practical implementation, a pointer to the start of each context is stored in a hash table. If this context appears, one follows the pointer and finds the correct continuation. Then, the pointer is incremented by one.*" "*(the last paragraph) At higher orders, an array representing all possible contexts may not be practical, and the simple array can be replaced by a trie, tree, hash-table, or other approach. Note that trees and tries can be walked in the order needed for AddCounters, whereas the contexts are additionally sorted for AddCounters if using a hash table.*"

```
INVERSE-ST(L_k, I)
    ▷ L_k is the last column vector
    ▷ I is the index of S_1
 1  len ← LENGTH(L_k), A[ ] ← ∅, B[ ] ← ∅
 2  for j ← 1 to k
 3      do
 4          for i ← 1 to len
 5              do
 6                  B[i] ← L_k[i] ⊕ A[i]
 7          A ← SORT(B)
 8  j ← I
 9  i ← len
10  while i > 0
11      do
12          S[i] ← L_k[j] ▷ restore the ith character.
            ▷ update the current context.
13          context ← [L_k[j], A[j, 1 : k − 1]]
            ▷ lookup the index of the immediate
            predecessor of the current character.
14          j ← LOOKUP(context)
            ▷ update the pointer for the current context.
15          UPDATE_POINTER(context)
16          i = i − 1
17  return S
```

Fig. 4. The inverse ST algorithm.

connection between the ST and the BWT from the algorithm perspective, despite the fact that the algorithm for the ST is not the most efficient algorithm for the BWT.

On the other hand, the existing algorithms of the two inverse transforms are dramatically different from each other. The inverse BWT has a succinct solution, but the inverse ST appears to be involved. What complicates the existing solutions for the inverse ST are 1) the k-order contexts retrieval and 2) the context lookup via the help of hash tables. The first factor implies $O(kN)$ time/space complexities. The second could be mathematically vague in hashing function design for any nontrivial k. Therefore, the inverse ST has not been solved satisfactorily and remains an open problem.

4 Unifying Two Transforms

4.1 Basis

In this section, we discuss several properties regarding the ST and prove several theorems accordingly.

Property 5. *For any text S ending with a sentinel \$, each k-order context in M_k is distinct when $k \in [N-1, N]$, that is, there are no two k-order contexts in M_k that are identical for $k \in [N-1, N]$.*

This property can be easily seen from the following observations on M_{N-1}: 1) There is only one context without \$ and 2) for all other contexts with \$, there is only one \$ in each context and its positions in any two contexts are different. Furthermore, given that all of the $(N-1)$-order contexts are different, it is easy to see that all of the corresponding N-order contexts are also different. This is because appending one more character to two different contexts, respectively, never makes them the same. This property implies that $M_{N-1} = M_N$.

Property 6. *For $k \in [2, N]$ and $i \in [1, N]$,*

$$M_k[T_k[i], 1:k] = L[i] \oplus M_k[i, 1:k-1]$$

for $k = 1$ and $i \in [1, N]$, $M_k[T_k[i], 1] = L[i]$.

This property comes directly from the definitions of M_k, L_k, and T_k.

Property 7. *For $k \in [1, N]$, all of the rows in M_k with the same first $h \in [1, k]$ characters are arranged consecutively.*

This property can be easily seen from the ST's lexicographical sorting criterion.

Property 8. *For $k \in [1, N]$, if $M_k[i, 1:k] = M_k[j, 1:k]$ and $1 \le i < j \le N$, $L_k[j]$ must be recovered before $L_k[i]$ unless i is I.*

This property is due to the ST's positional sorting criterion. The criterion states that if two rows share the same k-order context, their relative order in the original OM matrix must remain the same in M_k. Consequently, the relative order of the last characters of the two rows in OM must be preserved in M_k. Therefore, the two rows sharing the same k-order context must be visited in reverse order by the inverse ST algorithms because the restoration procedure under the discussion starts from the last character and dynamically proceeds in reverse order. The exception occurs when $i = I$; in this special case, $L_k[i]$, that is, the last character of the original text, will always be the first character to be recovered.

Properties 7 and 8 reflect the ST's two sorting criteria, respectively.

Property 9. *For $k \in [1, N]$ and $i \in [1, N-1]$, if there exists any $j \in [1, k]$, $M_k[i, j] \neq M_k[i+1, j]$, we have $M_k[i, 1:k] \neq M_k[i+1, 1:k]$.*

This property states that it is not necessary to wait until all of the first k columns have been completely restored to detect context switches between any two successive k-order contexts in M_k; instead, a context switch can be detected early whenever a difference existing at a certain column is encountered in the middle of the column by column context restoring procedure.

Property 10. $M_k[i, 1:k] \neq M_k[i+1, 1:k]$ *if*

1. $M_k[i, 1:k/2] \neq M_k[i+1, 1:k/2]$ *or*
2. $M_k[i, 1:k/2] = M_k[i+1, 1:k/2]$ *and*

$$M_k[i, k/2+1:k] \neq M_k[i+1, k/2+1:k],$$

where $i \in [1, N-1]$, $k \in [1, N]$, and $k/2$ is a positive integer.

This property can be easily reasoned as follows: If the k-order contexts of two consecutive rows are different, the difference must fall into one of the two cases: either 1) their

first halves are already different or 2) their first halves are identical but their second halves are different.

Property 11. *$M_k[i, k/2+1:k] \neq M_k[i+1, k/2+1:k]$ if*

$$M_k[Q^{k/2}[i], 1:k/2] \neq M_k[Q^{k/2}[i+1], 1:k/2],$$

where $i \in [1, N-1]$, $k \in [1, N]$, and $k/2$ is a positive integer, $Q^{k/2}[i]$ and $Q^{k/2}[i+1]$ locate the $k/2$th generation pseudosuccessors of the given ith and $(i+1)$th rows, respectively.

In order to detect whether the second halves of two k-order contexts are different, we can instead compare the first halves of their corresponding $k/2$ generation pseudosuccessors. Notice that $Q^{k/2}[i]$ depends on k and i, which can be computed by the recursive algorithm $QT(row, korder)$ given below. To get $Q^{k/2}[i]$, we call the function by providing parameters row and $korder$ with i and $k/2$, respectively.

QT$(row, korder)$
  if $korder = 1$
    then return $Q_k[row]$;
  return QT(**QT**$(row, korder/2), korder/2)$;

By combining Properties 10 and 11, we can quickly mark off the row positions where switches of the clustered contexts occur. For example, in order to mark off context switches for 16-order contexts, we need to mark off context switches for 8-order contexts first, which in turn needs to compare 4-order contexts, and so on.

Based on the above observations, we now prove several theorems as follows.

Theorem 1. *For any $L_k[i] = L_k[j]$ and*

$$M_k[T_k[i], 1:k] = M_k[T_k[j], 1:k],$$

where $k \in [1, N]$ and $1 \leq i < j \leq N$, we have $M_k[T_k[r], 1:k] = M_k[T_k[i], 1:k]$, for any $r \in [i, j]$ and $L_k[r] = L_k[i]$.

Proof.

- For $k = 1$, it is trivial to show that the result is correct because, in this case, $L_k[h] = M_k[T_k[h], 1]$, for $h \in [1, N]$.
- For $k \in [2, N]$, given that $L_k[i] = L_k[j]$, $i < j$, and $M_k[T_k[i], 1:k] = M_k[T_k[j], 1:k]$, from Property 6, we have $M_k[i, 1:k-1] = M_k[j, 1:k-1]$. Given that $i < r < j$ and

  $$M_k[i, 1:k-1] = M_k[j, 1:k-1],$$

  according to Property 7, we have

  $$M_k[i, 1:k-1] = M_k[r, 1:k-1] = M_k[j, 1:k-1].$$

  Further, given that $L_k[r] = L_k[i]$, again, from Property 6, we have

  $$M_k[T_k[i], 1:k] = L[i] \oplus M_k[i, 1:k-1],$$
  $$M_k[T_k[r], 1:k] = L[r] \oplus M_k[r, 1:k-1],$$

  and $M_k[T_k[j], 1:k] = L[j] \oplus M_k[j, 1:k-1]$. □

This theorem states that, for any two identical characters $L_k[i]$ and $L_k[j]$ in L_k, if the corresponding rows in M_k starting with the character also have the same first k characters, that is, $M_k[T_k[i], 1:k] = M_k[T_k[j], 1:k]$, then, for any character $L_k[r] = L_k[i]$ in L_k located between $L_k[i]$ and $L_k[j]$, the corresponding row of M_k starting with the character $L_k[r]$ also has the same first k characters as $M_k[T_k[i], 1:k]$, that is, $M_k[T_k[r], 1:k] = M_k[T_k[i], 1:k]$.

Theorem 2. *Let A_k be a k-by-N matrix satisfying 1) $A_k[1, P_k[j]] = L_k[j]$ and 2) $A_k[i, P_k[j]] = A_k[i-1, j]$, for $i \in [2, k]$, where $k \in [1, N]$ and $j \in [1, N]$, then $A_k = M_k[1:N, 1:k]^T$.*

Proof. Let $B_k = \begin{bmatrix} L_k \\ A_k \end{bmatrix}$. According to condition 1) and the definition of P_k, the elements of $A_k[1, 1:N]$, from left to right, store the characters of L_k in their lexicographical orders. The columns of $B_k[1:2, 1:N]$ give all of the 2-order contexts of M_k. Given that all of the columns of $A_k[1, 1:N]$ have been sorted, from condition 2), it is easy to see that the formula $A_k[2, P_k[j]] = A_k[1, j]$ is equivalent to sorting all of the columns of $B_k[1:2, 1:n]$ in their lexicographical orders into $A_k[1:2, 1:N]$, with $L_k[j]$ being the most significant character for each column j. The columns of the resulting $A_k[1:2, 1:N]$ give all of the 2-order contexts of M_k sorted in their lexicographical orders. The same argument applies to $A_k[i, 1:N]$ for $i \in [3, k]$. To generalize, for any $i \in [2, k]$, given that all of the columns of $A_k[1:i-1, 1:N]$ have been sorted, $A_k[i, P_k[j]] = A_k[i-1, j]$ will result in all the columns of $A_k[1:i, 1:N]$ being sorted lexicographically with the most significant character of each column $A_k[1:i, j]$ being $A_k[1, j]$, for $j \in [1, N]$, and the columns of $A_k[1:i, 1:N]$ give all of the i-order contexts of M_k sorted in their lexicographical orders. As a result, the columns of A, from left to right, will recover all of the k-order contexts in their sorted order as $A_k = M_k[1:N, 1:k]^T$, for any $k \in [1, N]$. □

Theorem 3. *$T_k = P$ and each element of C_k is one when $k \in [N-1, N]$.*

Proof. There are no two identical k-order contexts when $k \in [N-1, N]$ (see Property 5); therefore, all elements of T_k are different and it turns out that T_k is the same as P (see the definition of P). At the same time, all elements of C_k are 1, indicating that each context appears exactly once.□

4.2 The Proposed Algorithms

Based on these properties and theorems, we propose three fast inverse ST algorithms. The first one is a novel auxiliary-vector-based solution that eliminates the need for using any hash table. Both the space and the time complexities of the solution remain $O(kN)$ because it still requires the restoration of M_k. The second eliminates the restoration of matrix M_k, thus reducing the space complexity from $O(kN)$ to $O(N)$. The third further reduces the time complexity from $O(kN)$ to $O(N \log_2 k)$.

4.2.1 Eliminating the Use of Hash Tables

The primary goal of the first algorithm is to eliminate the use of hash tables while retaining the ability to correctly locate the

```
GIBWT-M(L_k, I, k)
    ▷ L_k is the last column vector
    ▷ I is the row index of S_1
    ▷ k is the order of the ST
 1  len ← LENGTH(L_k), POPULATE(L_k, P_k, Q_k)
 2  RESTORE(M_k[1 : len, 1 : k])
 3  INITIALIZE_COUNTER(C_k[ ])
    ▷ calculate the frequency and the starting position of
    each context
 4  for i ← 1 to len
 5       do
 6          if i = 1 or M_k[i − 1, 1 : k] ≠ M_k[i, 1 : k]
 7             then j ← i
 8          T_k[Q_k[i]] ← j
 9          C_k[j] ← C_k[j] + 1
    ▷ restore S
10  j ← I
11  i ← len
12  while i > 0
13       do
14          S[i] ← L_k[j] ▷ restore the ith character
15          j ← T_k[j]
16          C_k[j] ← C_k[j] − 1
            ▷ update the indexes for the next character
17          j ← j + C_k[j]
18          i ← i − 1
19  return S
```

Fig. 5. The GIBWT-NM algorithm (Generalized Inverse BWT with M_k).

true immediate predecessor row for any given row. For this purpose, we introduce two vectors, T_k and C_k, through which we can transform the context lookup problem into a vector lookup problem. This algorithm is named GIBWT-M since we deem the inverse ST to be a generalized inverse BWT and the algorithm relies on the complete restoration of M_k. The algorithm is shown in Fig. 5.

The algorithm consists of four modules. In the first module, the characters of L_k are sorted to obtain F_k; then P_k and Q_k are computed. Following the notations and definitions in Section 2, we acknowledge that Q_k is simply the inverse vector of P_k, that is, Q_k and P_k are two mapping vectors between L_k and F_k reciprocal to each other. Therefore, Q_k and P_k can be calculated at the same time ($i = Q_k[P_k[i]]$) by simply traversing L_k and F_k once.

In the second module, L_k, I, and P_k are used to restore the k-order context matrix $M_k[1 : N, 1 : k]$. Since how to restore $M_k[1 : N, 1 : k]$ is not the main concern of our algorithm, it is represented by a subfunction call at line 2 in Fig. 5.

In the third module, C_k and T_k are calculated. According to Section 2, T_k is an index vector pointing to the starting row for each unique k-order context and C_k is a counter vector recording the occurrence of each unique k-order context. From Property 7, we know that all of the rows of M_k with the same first k characters, that is, the same k-order contexts, are clustered together. This property suggests that, in order to populate C_k, we can scan through all k-order contexts of M_k and count the frequency of each context in one single pass. In the same pass, we can also compute T_k based on Theorem 1 and the definition of Q_k. In our algorithm, the logic to simply traverse every pair of neighbor rows of the restored k-order context matrix $M_k[1 : N, 1 : k]$ once for calculating T_k and C_k is coded by the lines from 3 to 9. As the result of the conditional statement of updating j (see lines 6 and 7), line 8 will update an element of T_k with a new value only when a new context is found and line 9 will accumulate C_k only at the index where a context is first found.

Once T_k and C_k are calculated, the fourth module will use them to restore the original text S without looking up any hash table. Specifically, given a row, the true index for its immediate predecessor can be deduced from the static T_k and the dynamic C_k. T_k is static because it is calculated only once (see Property 7) and will be fixed throughout the running of the algorithm. C_k is dynamic because, whenever an element of C_k is used, the element will decrease by 1 during the runtime of the algorithm.

The logic of restoring S starting from $L_k[I]$ is coded by the lines from 10 to 18. It should be noted that lines 14 to 18 perfectly reflect Property 8 and the exception of Property 8 is enforced by lines 10 and 14. At the end, the procedure returns the recovered S at line 19.

4.2.2 Eliminating the Restoration of M_k

In the GIBWT-M algorithm (see Fig. 5), we have assumed complete knowledge of $M_k[1 : N, 1 : k]$, which can be reconstructed using Theorem 2. Once $M_k[1 : N, 1 : k]$ is completely retrieved, it is straightforward to compute C_k and T_k by simply comparing every two consecutive k-order contexts to check whether they differ from each other. However, once C_k and T_k have been computed, M_k will be useless in the rest of the algorithm. Therefore, in our framework, the text retrieval relies on C_k and T_k directly and $M_k[1 : N, 1 : k]$ indirectly.

Obviously, $M_k[1 : N, 1 : k]$ consumes a $k \times N$ matrix space, while the sole purpose of completely restoring it is to compute T_k and C_k. Inspired by this observation, we propose a more space-efficient alternative to GIBWT-M by taking advantage of Property 9. This property states that, for the purpose of context switch detection, we actually do not need to allocate a static $k \times N$ matrix space for storing all of the complete k-order contexts; rather, we only need to retrieve the k-order contexts column by column on the fly by comparing every pair of neighbor rows only at that column for context switches. This way, we only need a space of $2N$ to alternate the current column with the new column (actually, the next column derived from the current column) to achieve the goal.

This algorithm is named GIBWT-NM (Generalized Inverse BWT without M_k) and is shown in Fig. 6. The GIBWT-NM algorithm utilizes a column-wise difference detection scheme to directly find out where context switches occur without concatenating all the characters of each k-order context. Specifically, this new algorithm does not produce $M_k[1 : N, 1 : k]$ but uses a size-N vector D to record context differences (see the lines 1 to 19). The D vector can be obtained by comparing G_0 and G_1 and can

```
GIBWT-NM(L_k, I, k)
    ▷ L_k is the last column vector
    ▷ I is the index of S_1
    ▷ k is the order of ST
 1  len ← LENGTH(L_k), POPULATE(L_k, P_k, Q_k)
 2  for i ← 1 to len
 3       do
 4          C_k[i] ← 0 ▷ initialize the counter array to 0.
 5          G_0[i] ← L_k[i]
 6          D[i] ← 0
 7  odd ← 1, D[1] ← 1
 8  for j ← 1 to k
 9       do
10          for i ← 1 to len
11               do
12                  if odd = 1
13                    then G_1[P_k[i]] ← G_0[i]
14                    else G_0[P_k[i]] ← G_1[i]
15          odd ← 1 − odd
16          for i ← 2 to len
17               do
18                  if (odd = 1 and G_0[i] ≠ G_0[i − 1]) or
                       (odd = 0 and G_1[i] ≠ G_1[i − 1])
19                    then D[i] ← 1
20  for i ← 1 to len
21       do
22          if D[i] = 1
23            then j ← i
24          T_k[Q_k[i]] ← j
25          C_k[j] ← C_k[j] + 1
26  Call the same S restoring procedure in GIBWT-M.
27  return S
```

Fig. 6. The GIBWT-NM algorithm (Generalized Inverse BWT without M_k).

help compute the starting point of each context without using $M_k[1:N, 1:k]$ at all (see lines 20 to 25 for the calculation of T_k and C_k). Once T_k and D_k are available, line 26 calls the same subroutine that is used in algorithm GIBWT-M (see lines 10 to 18 in Fig. 5) to restore S.

Compared with algorithm GIBWT-M, the space complexity for the GIBWT-NM algorithm has been reduced to $O(2N) = O(N)$, which makes it more hardware-implementation friendly. Actually, a counter can be used to keep track of how many rows have been detected as being different from their neighbors. If the counter reaches $N-1$, we can terminate the difference detection process even earlier. However, in the worst case, up to k columns will still be retrieved on the fly. Therefore, from the algorithm analysis perspective, although the space complexity has been dramatically reduced from $O(kN)$ to $O(N)$, the time complexity remains $O(kN)$ (for a detailed complexity analysis, see Section 4.3).

4.2.3 A Faster Inverse ST Algorithm

Our third algorithm, which is called GIBWT-NML (Generalized Inverse BWT without M_k and with $Logk$ speedup) and is described in Fig. 7, shows that the time complexity of the algorithm GIBWT-NM can be further reduced to $O(N \log_2 k)$.

```
GIBWT-NML(L_k, I, k)
    ▷ L_k is the last column vector
    ▷ I is the index of S_1
    ▷ k is the order of ST
 1  len ← LENGTH(L_k), POPULATE(F_k, P_k, Q_k)
 2  for i ← 1 to len
 3       do
 4          C_k[i] ← 0 ▷ initialize the counter array to 0.
 5          P[i] ← P_k[i]
 6          D_0[i] ← F_k[i]
    ▷ Calculating the number and the start position of
    each k-order context.
 7  odd ← 1
 8  for j ← 1 to log_2 k
 9       do
10          for i ← 1 to len
11               do
12                  if (odd = 1)
13                    then
14                         H[i] ← P[P[i]]
15                         D_1[P[i]] ← D_0[i]
16                    else
17                         P[i] ← H[H[i]]
18                         D_1[H[i]] ← D_0[i]
19          for i ← 1 to len
20               do
21                  if (i = 1) or (D_0[i] ≠ LastD0) or
                       (D_1[i] ≠ D_1[i − 1])
22                    then
23                         LastD0 ← D_0[i]
24                         w ← i
25                  D_0[i] = w
26          odd ← 1 − odd
27  for i ← 1 to len
28       do
29          if (i = 1) or (D_0[i] ≠ D_0[i − 1])
30            then
31                 j ← i
32          T_k[Q_k[i]] ← j
33          C_k[j] ← C_k[j] + 1
34  Call the same S restoring procedure in GIBWT-M.
35  return S
```

Fig. 7. The GIBWT-NML algorithm (Generalized Inverse BWT without M_k with $\log_2 k$ speedup).

The algorithm is built upon Properties 10 and 11. The idea behind the two properties is to double the steps to reach column k without going through every column so that the context switch detection time can be greatly reduced.

Properties 10 and 11 suggest a recursive implementation of the context switch detection. However, considering that a row will be compared with both its upper and lower neighbor rows, a naive recursive implementation solving the common subproblems repeatedly at deep levels will do more work than necessary in this context. In our algorithm design, we adopt an efficient dynamic programming technique that solves every subproblem bottom up just once and then saves its answer in vectors for the next jump. Specifically, we will

iteratively perform comparisons for columns 1, 2, 4, and so on. Our solution avoids the work of repeating context switch detection every time the subproblem is encountered. This "binary search" style operation requires approximately $\log_2 k$ visits for k-order contexts because each step of the algorithm reduces the part of the context that we continue to compare by a factor of 2, resulting in the function of log to the base 2 in the complexity. As a result, the GIBWT-NML algorithm uses the so-called "doubling technique" to accelerate context switch detection and dramatically reduce the runtime complexity from $O(kN)$ to $O(N \log_2 k)$.

4.2.4 Generalizing the Inverse BWT

Our algorithms can better explain the relationship between the inverse ST and the inverse BWT because the algorithms can actually be used to invert both. For example, the algorithm GIBWT-M, originally designed for computing the inverse ST, will develop into a solution for the inverse BWT when $k \in [N-1, N]$. In this sense, we have provided a unified solution (GIBWT-M or its variants) for both the inverse ST and the inverse BWT.

Corollary 1. *Algorithm GIBWT-M for the inverse ST will also be a solution for the inverse BWT when $k \in [N-1, N]$.*

Proof. From the GIBWT-M algorithm and the proof of Theorem 3, we know that, when $k \in [N-1, N]$, line 16 will change the current element of C_k to 0 and every element of C_k will be visited only once, since every element in T_k is unique. Hence, C_k plays no function at all, which implies that only T_k is used to locate the next index (see lines 15 and 17 in Fig. 5) at each iteration. Considering that $T_k = P$ when $k \in [N-1, N]$ (see Theorem 3) and comparing Algorithm GIBWT-M (see Fig. 5) with the inverse BWT algorithm (see Fig. 2), the corollary is thus proven. □

It should be noted that, if $k \geq N-1$, the traditional inverse BWT algorithm can be directly used to solve the problem; however, when $k < N-1$, the GIBWT-NML algorithm, to the best of our knowledge, is the most efficient solution currently available to invert ST.

4.3 Complexity Analysis

Since each of the three proposed efficient inverse ST algorithms consists of three or four functional modules, we analyze the time and the space complexities of each constituent module first.

- *Module 1*. Constructing the vectors P_k and Q_k by traversing L_k two passes at most. This requires a time/space complexity of $O(N)$.
- *Module 2*. Completely reconstructing all of the k-order contexts. This requires a time/space complexity of $O(kN)$.
- *Module 3*. Computing the vectors T_k and C_k. This module is performed differently in different algorithms. In algorithm GIBWT-M, this step relies on the fully restored k-order contexts (see Module 2), thus requiring a time/space complexity of $O(kN)$. In algorithm GIBWT-NM, this step skips module 2 but reuses two vectors, G_0 and G_1, thereby reducing the space complexity to $O(N)$, which is independent of the values of k. However, this step still visits up to k columns on the fly in the worst case, so the time complexity of this step remains unchanged as $O(kN)$. In algorithm GIBWT-NML, which is an improvement of GIBWT-NM, we can further reduce the time complexity to $O(N \log_2 k)$ by visiting $\log_2 k$ columns.
- *Module 4*. Restoring the original text S from S^{st-k}. This particular step relies solely on the two auxiliary vectors of T_k and C_k. Once T_k and C_k have been prepared by Module 3, this recovering procedure (described in the algorithm GIBWT-M) can be conducted in a time/space complexity of $O(N)$. Although module 3 can be performed differently, module 4 will be finally invoked by all three algorithms (see the last loop of the algorithm GIBWT-M that will be called by GIBWT-NM and GIBWT-NML).

Given that the constituent modules of each algorithm will be executed sequentially, we can easily find the time complexity of each algorithm by considering all of the time complexities of the participating modules. The space complexity can also be derived similarly. Depending on how module 3 is conducted and whether module 2 is involved or not (note that modules 1 and 4 are independent of any specific algorithm), the time/space complexities for each individual algorithm are analyzed as follows: The algorithm GIBWT-M, consisting of all four modules, demands a total time/space complexity of $O(kN)$. The algorithm GIBWT-NM, consisting of modules 1, 3, and 4, reduces the space complexity to $O(N)$ but keeps the space complexity as $O(kN)$ of GIBWT-M. The algorithm GIBWT-NML, also consisting of modules 1, 3, and 4, further reduces the time complexity to $O(N \log_2 k)$ and keeps the space complexity as $O(N)$ of GIBWT-NM.

In some text data sets, k can be chosen to be a number far less than N; therefore, both the time and the space complexities of our algorithms can be regarded as being $O(N)$. However, k should be kept in the big-O notations because, from both the theory and engineering points of view, the choice of k matters, especially when k becomes too large to be negligible. This could happen in certain applications where highly redundant data is of concern or when memory-critical hardware implementation is under consideration. In such situations, our third algorithm has a time complexity of $O(N \log_2 k)$ and a space complexity $O(N)$, which far outperforms the time/space complexities of $O(kN)$ required by the traditional inverse ST algorithms.

4.4 Discussion

The backward text retrieval strategy of the inverse ST and the inverse BWT is actually an iterative process to locate the preceding character of a given character by deducing the immediate predecessor row from the row which ends with the given character. For the inverse ST, Schindler suggested a hashing approach to find out the immediate predecessor row without providing any mathematical explanation of the method [23]. Later, Yokoo provided a more thorough mathematical reasoning for the inverse ST and proposed a nonhashing approach [13]. Yokoo's method can locate the correct predecessor row for a given row by dynamically crossing off the newly recovered row; however, it still relies on the complete restoration of all k-order contexts and it does *not* use any auxiliary vectors as we do. As a contrast, our algorithms use the two auxiliary vectors T_k and C_k, not

M_k, to retrieve the original text. The two auxiliary vectors are the signature data structures of our algorithms which distinguish our algorithms from the solutions proposed by Schindler and Yokoo.

Further, the three algorithms can be differentiated from each other by the ways in which they calculate the two auxiliary vectors. In our first algorithm, GIBWT-M, we decompose the original k-order contexts restoration phase used by Schindler and Yokoo into two phases. The first phase still restores all of the k-order contexts, which will be used in the second phase to calculate the two auxiliary vectors T_k and C_k. These two auxiliary vectors will facilitate the text retrieval through a nonhashing approach. In our second algorithm, GIBWT-NM, we replace the k-order context retrieval by calculating a context switch flag vector D, through which the two auxiliary vectors T_k and C_k can be correctly derived without using static k-order contexts. In order to calculate the vector D, we still visit every column in the worst case; however, we do not need to save every column in memory; thus, the second algorithm improves the space complexity, but does not improve the time complexity. In our third algorithm, GIBWT-NML, we further improve the runtime complexity by using the "doubling technique" to quickly detect any two different consecutive k-order contexts. The "doubling technique" has been a common practice used in fast suffix sorting algorithms to reorder context pointers [14], [31]; however, it is the *first* time for the "doubling technique" to be used to accelerate the process of context switch detection in the inverse ST algorithms.

4.5 An Example

We use a running example for the 4-order inverse ST to illustrate the key contributions of our algorithms: 1) how to efficiently calculate the auxiliary vectors D, T_k, and C_k and 2) how to use them to retrieve the original text S from L_k.

Given $k = 4$, the ST will transform $S = [m\ \ i\ \ s\ \ s\ \ i\ \ s\ \ s\ \ i\ \ p\ \ p\ \ i\ \ \$]$ into the matrix M_4 shown in Fig. 3, from which we have

$$L_k = [s\ \ m\ \ s\ \ p\ \ \$\ \ p\ \ i\ \ s\ \ s\ \ i\ \ i\ \ i],$$

and $I = 5$.

Now, provided with L_k and I, the inverse ST starts with calculating F_k. By simply sorting all of the characters of L_k, we obtain $F_k = [i\ \ i\ \ i\ \ i\ \ m\ \ p\ \ p\ \ s\ \ s\ \ s\ \ s\ \ \$]$.

From L_k and F_k, we can easily calculate P_k and Q_k as

$$\begin{array}{rcl} P_k & = & [8\ \ 5\ \ 9\ \ 6\ \ 12\ \ 7\ \ 1\ \ 10\ \ 11\ \ 2\ \ 3\ \ 4], \\ Q_k & = & [7\ \ 10\ \ 11\ \ 12\ \ 2\ \ 4\ \ 6\ \ 1\ \ 3\ \ 8\ \ 9\ \ 5]. \end{array}$$

Please note that neither P_k nor Q_k can be directly used to retrieve the immediate predecessor or successor row for a given row. For example, P_k shows that $P_k[10] = 2$, but the predecessor of the 10th row is actually the third row, instead of the second row.

Next, we can calculate the vector D by the algorithm GIBWT-NML and then compute T_k and C_k from D as

$$\begin{array}{rcl} D & = & [1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1], \\ T_k & = & [8\ \ 5\ \ 9\ \ 6\ \ 12\ \ 7\ \ 1\ \ 10\ \ 11\ \ 2\ \ 2\ \ 4], \\ C_k & = & [1\ \ 2\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1]. \end{array}$$

To illustrate how to use T_k and C_k, we examine the values of $T_k[10]$, $T_k[11]$, and the correlated $C_k[2]$. The element $T_k[10] = 2$ indicates that the immediate predecessor of the 10th row is not necessarily to be the second row (actually, it is not in this case), but a row sharing the same k-order context with the second row. In fact, the value of 2 is simply the starting row index of the duplicated k-order contexts. Similarly, $T_k[11] = 2$ conveys exactly the same meaning. Correspondingly, $C_k[2] = 2$ shows that the k-order context of the second row repeats at two consecutive rows, starting from the second row. Combining the above information, it means that a new k-order context starts from the second row and repeats itself at the two consecutive rows (second and third). Given that the vectors C_k and T_k have been prepared, the last stage of GIBWT-NML can easily restore the original text S from L_k, one character per iteration.

$S = [\ ]$; $j = I$;
i=12: $S[i] = L_k[j] =$'$'; $j = T_k[j] = 12$;
$C_k[j] = C_k[j] - 1 = 1 - 1 = 0$; $j = j + C_k[j] = 12$;
i=11: $S[11] = L_k[12] =$'i'; $j = T_k[12] = 4$;
$C_k[4] = C_k[4] - 1 = 0$; $j = 4 + C_k[4] = 4$;
i=10: $S[10] = L_k[4] =$'p'; $j = T_k[4] = 6$;
$C_k[6] = C_k[6] - 1 = 0$; $j = 6 + C_k[6] = 6$;
i=9: $S[9] = L_k[6] =$'p'; $j = T_k[6] = 7$;
$C_k[7] = C_k[7] - 1 = 0$; $j = 7 + C_k[7] = 7$;
i=8: $S[8] = L_k[7] =$'i'; $j = T_k[7] = 1$;
$C_k[1] = C_k[1] - 1 = 0$; $j = 1 + C_k[1] = 1$;
i=7: $S[7] = L_k[1] =$'s'; $j = T_k[1] = 8$;
$C_k[8] = C_k[8] - 1 = 0$; $j = 8 + C_k[8] = 8$;
i=6: $S[6] = L_k[8] =$'s'; $j = T_k[8] = 10$;
$C_k[10] = C_k[10] - 1 = 0$; $j = 10 + C_k[10] = 10$;
i=5: $S[5] = L_k[10] =$'i'; $j = T_k[10] = 2$;
$C_k[2] = C_k[2] - 1 = 1$; $j = 2 + C_k[2] = 3$;
i=4: $S[4] = L_k[3] =$'s'; $j = T_k[3] = 9$;
$C_k[9] = C_k[9] - 1 = 0$; $j = 9 + C_k[9] = 9$;
i=3: $S[3] = L_k[9] =$'s'; $j = T_k[9] = 11$;
$C_k[11] = C_k[11] - 1 = 0$; $j = 11 + C_k[11] = 11$;
i=2: $S[2] = L_k[11] =$'i'; $j = T_k[11] = 2$;
$C_k[2] = C_k[2] - 1 = 0$; $j = 2 + C_k[2] = 2$;
i=1: $S[1] = L_k[2] =$'m'; $j = T_k[2] = 5$;
$C_k[5] = C_k[5] - 1 = 0$; $j = 5 + C_k[5] = 5$;

Fig. 8. A running example for the text retrieval procedure in GIBWT-M.

Following the running example of the character-by-character text retrieval in Fig. 8, we finally obtain $S = [m\ \ i\ \ s\ \ s\ \ i\ \ s\ \ s\ \ i\ \ p\ \ p\ \ i\ \ \$]$, which is exactly the original input text to the ST. Notice that, at the end of the last iteration when the original text S has been completely restored, j returns to 5, the value of the given index I. This is exactly what a correct retrieval procedure should have done, that is, gone through a perfect cycle.

5 Conclusion

In [23], Schindler proposed the concept of limited-order and unlimited-order contexts to extend the original BWT [1] to the ST, which can run much faster than the BWT by limiting the context order to a small value. However, the existing inverse ST solutions proposed in [13], [23], [24] rely on the complete restoration of all k-order contexts and employ hash tables to look up contexts. Such inverse ST approaches are dramatically different in nature from the way in which the inverse BWT works.

In this paper, we have presented a set of properties and theorems that can better explain the two transforms. Based

on these properties and theorems, we propose a novel auxiliary-vector-based framework that leads to three fast and memory-efficient inverse ST algorithms. With these algorithms, we show that the BWT and the ST, as well as their respective inverse components, can be conducted under the same framework with different context order settings: N for the BWT and $[1, N]$ for the ST, respectively. This means that, when k reaches N, the inverse ST will automatically become the inverse BWT and the ST will automatically regress to the BWT as well. Therefore, from the algorithm point of view, we have clearly revealed the strong connection between the ST and the BWT, as well as that between the inverse ST and the inverse BWT.

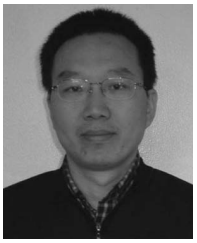
ACKNOWLEDGMENTS

The authors wish to thank the *IEEE Transactions on Computers* anonymous reviewers and the editors for their constructive suggestions and insightful comments that have helped enrich the content and improve the presentation of this paper. The authors thank Kathleen Meeker for correcting and proofreading the manuscript. The work of Ge Nong is supported in part by GuangDong Natural Science Foundation research grant number 06023193.

REFERENCES

[1] M. Burrows and D.J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," SRC Research Report 124, Digital Systems Research Center, Calif., May 1994.
[2] D. Adjeroh et al., "DNA Sequence Compression Using the Burrows-Wheeler Transform," *Proc. IEEE CS Bioinformatics Conf.*, pp. 303-313, Aug. 2002.
[3] Z. Arnavut, "Lossless and Near-Lossless Compression of ECG Signals," *Proc. 23rd Ann. Int'l Conf. IEEE Eng. in Medicine and Biology Soc.*, vol. 3, pp. 2146-2149, Oct. 2001.
[4] A. Mukherjee et al., "Prototyping of Efficient Hardware Algorithms for Data Compression in Future Communication Systems," *Proc. 12th Int'l Workshop Rapid System Prototyping (RSP '01)*, pp. 58-63, June 2001.
[5] M. Schindler and B. Sebastian, "Image Compression Using Blocksort," *Proc. Data Compression Conf. (DCC '01)*, p. 515, Mar. 2001.
[6] Z. Arnavut, "Lossless Compression of Color-Mapped Images," *Optical Eng.*, vol. 38, no. 6, pp. 1001-1005, June 1999.
[7] B. Balkenhol and S. Kurtz, "Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice," *IEEE Trans. Computers*, vol. 49, no. 10, pp. 1043-1053, Oct. 2000.
[8] B. Balkenhol, S. Kurtz, and Y.M. Shtarkov, "Modifications of the Burrows and Wheeler Data Compression Algorithm," *Proc. Data Compression Conf. (DCC '99)*, pp. 188-197, Mar. 1999.
[9] G. Manzini, "The Burrows-Wheeler Transform: Theory and Practice," *Proc. 24th Int'l Symp. Math. Foundations of Computer Science (MFCS '99)*, pp. 34-47, Sept. 1999.
[10] G. Manzini, "An Analysis of the Burrows-Wheeler Transform," *J. ACM*, vol. 48, no. 3, pp. 407-430, May 2001.
[11] Z. Arnavut and M. Arnavut, "Investigation of Block-Sorting of Multiset Permutations," *Int'l J. Computer Math.*, vol. 81, no. 10, pp. 1213-1222, Oct. 2004.
[12] Z. Arnavut, "Generalization of the BWT Transformation and Inversion Ranks," *Proc. Data Compression Conf. (DCC '02)*, p. 447, Apr. 2002.
[13] H. Yokoo, "Notes on Block-Sorting Data Compression," *Electronics and Comm. in Japan (Part III: Fundamental Electronic Science)*, vol. 82, no. 6, pp. 18-25, 1999.
[14] K. Sadakane, "A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation," *Proc. Data Compression Conf. (DCC '98)*, pp. 129-138, Mar. 1998.
[15] D. Baron and Y. Bresler, "Antisequential Suffix Sorting for BWT-Based Data Compression," *IEEE Trans. Computers*, vol. 54, no. 4, pp. 385-397, Apr. 2005.
[16] J. Karkkainen and P. Sanders, "Simple Linear Work Suffix Array Construction," *Proc. 30th Int'l Colloquium on Automata, Languages, and Programming (ICALP '03)*, pp. 943-955, 2003.
[17] D.K. Kim, J.S. Sim, H. Park, and K. Park, "Linear-Time Construction of Suffix Arrays," *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, pp. 186-199, 2003.
[18] P. Ko and S. Aluru, "Space Efficient Linear Time Construction of Suffix Arrays," *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, pp. 200-210, 2003.
[19] N.J. Larsson, "The Context Trees of Block Sorting Compression," *Proc. Data Compression Conf. (DCC '98)*, pp. 189-198, Mar. 1998.
[20] R.M. Karp, R.E. Miller, and A.L. Rosenberg, "Rapid Identification of Repeated Patterns in Strings," *Proc. Fourth ACM Symp. Theory of Computing*, pp. 125-136, 1972.
[21] F. Hongo and H. Yokoo, "Block-Sorting Data Compression and KMR Algorithm," *Proc. 20th Symp. Information Theory and Its Applications*, pp. 673-676, 1997.
[22] S.J. Puglisi, W.F. Smyth, and A. Turpin, "The Performance of Linear Time Suffix Sorting Algorithms," *Proc. Data Compression Conf. (DCC '05)*, pp. 358-367, Mar. 2005.
[23] M. Schindler, "A Fast Block-Sorting Algorithm for Lossless Data Compression," *Proc. Data Compression Conf. (DCC '97)*, p. 469, Mar. 1997.
[24] M. Schindler, "Method and Apparatus for Sorting Data Blocks," US patent 6,199,064, Mar. 2001.
[25] M. Schindler, Szip Homepage, http://www.compressconsult.com/szip/, 2007.
[26] I. Witten and T. Bell, Calgary Text Compression Corpus, ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/, 2007.
[27] T.B.R. Arnold, "A Corpus for the Evaluation of Lossless Compression Algorithms," *Proc. Data Compression Conf. (DCC '97)*, pp. 201-210, http://corpus.canterbury.ac.nz, Mar. 1997.
[28] G. Nong and S. Zhang, "Unifying the Burrows-Wheeler and the Schindler Transforms," *Proc. Data Compression Conf. (DCC '06)*, p. 464, Mar. 2006.
[29] G. Nong and S. Zhang, "An Efficient Algorithm for the Inverse ST Problem," *Proc. Data Compression Conf. (DCC '07)*, p. 397, Mar. 2007.
[30] S. Deorowicz, "Second Step Algorithms in the Burrows-Wheeler Compression Algorithm," *Software—Practice and Experience*, vol. 32, no. 2, pp. 99-111, Feb. 2002.
[31] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *Proc. First ACM-SIAM Symp. Discrete Algorithms*, pp. 319-327, 1990.

Ge Nong received the BE degree in computer engineering from the NanJing University of Aeronautics and Astronautics in 1992, the ME degree in computer engineering from the South China University of Science and Technology in 1995, and the PhD degree in computer science from the Hong Kong University of Science and Technology in 1999. After that, he joined STMicroelectronics as a researcher with R&D on integrated circuit and system technologies for high-speed switches and routers. Since July 2004, he has been an associate professor in the Computer Science Department at Sun Yat-Sen University, Guangzhou, People's Republic of China. His current research interests include algorithms, computer and communication networks, switching theory, and performance evaluation. He is a member of the IEEE.



Sen Zhang received the BS degree in computer science from TianJin University, TianJin, China, in 1992, the ME degree in computer engineering from the South China University of Science and Technology, Guangzhou, China, in 1995, and the PhD degree in computer science from the New Jersey Institute of Technology, Newark, in 2004. He joined the Department of Mathematics, Computer Science, and Statistics at the State University of New York, College at Oneonta in 2004, as an assistant professor. His current research interests include algorithms, data mining, database management, and bioinformatics. He is a member of the IEEE.