Extension and Faster Implementation of the GRP Transform for Lossless Compression

Hidetoshi Yokoo

Department of Computer Science, Gunma University
Kiryu 376-8515, Japan
yokoo@cs.gunma-u.ac.jp

Abstract. The GRP transform, or the *generalized radix permutation* transform was proposed as a parametric generalization of the BWT of the block-sorting data compression algorithm. This paper develops its extension that can be applied with any combination of parameters. By using the technique developed for linear time/space implementation of the sort transform, we propose an efficient implementation for the inverse transformation of the GRP transform. It works for arbitrary parameter values, and can convert the transformed string to the original string in time linear in the string length.

1 Introduction

The GRP transform, or the *generalized radix permutation* transform, was proposed by Inagaki, Tomizawa, and the present author in [3] as a parametric generalization of the BWT of the block-sorting data compression algorithm [1],[2]. The BWT and its variations [6],[7],[8] can be derived from the GRP transform as its special cases. The GRP transform has two parameters: the *block length* ℓ and the *context order* d, to which we can assign appropriate values so that we can also realize new transforms. In this sense, the GRP transform is a proper extension of those existing transforms. Preliminary experiments [4] show that some files are more efficiently compressed by an appropriate combination of parameter values with a tuned second-step encoder than by the original BWT.

In spite of its generality, the GRP transform is given concrete procedures only in the case where its parameters satisfy $n = b\ell$ and $0 \le d \le \ell$ for the length n of the string to be compressed and for an integer b. It is conceptually possible to remove these restrictions and to allow n, ℓ and d to be any natural numbers. However, it is not obvious whether we can run such a general version in an efficient way. In our previous paper [3], we have shown that the GRP transform can be performed in $O(n + bd) = O(n + nd/\ell)$ time. This implies that the transform runs in time linear in the string length, as long as the parameter d stays in the range of $0 \le d \le \ell$. However, it may require quadratic time when we wish to increase d beyond the range.

In the GRP transform, the inverse transformation is more complicated than the forward transformation. We observe a quite similar situation [5], [6] in the

Sort Transform (ST) [7], which has been proposed as a finite-order variant of the BWT. The ST, which is also a special case of the GRP transform, was originally developed to speed up the BWT. This aim was achieved in its forward transformation with a trade-off of a more demanding and complicated inverse transformation. Nong and Zhang [5] have addressed the problem of developing an efficient inverse ST transform, and gave a linear time/space algorithm with Chan in their recent paper [6].

In this paper, we show that the method developed by Nong, Zhang, and Chan [6] can be applied to the GRP transform so that its inverse transformation can be performed in linear time for any combination of the parameters. In order to show this and make clear the relation between the GRP transform and the BWT, we adopt a completely different description on the GRP transform than that given in our previous paper [3]. For example, in [3], the original data string to be compressed is arranged as a column vector of a matrix. In the present paper, on the other hand, we adopt the convention of the BWT, in which the original string is arranged as a row vector of a matrix. The BWT gathers those symbols that occur in the same or similar contexts in a source string, where the contexts are *backward* ones in the sense that the string is traversed from right to left. In this respect, too, we follow the BWT. The introduction of the conventions of the BWT and the ST to the description here makes it easy to understand that its inverse transformation runs in linear time.

The rest of the paper is organized as follows: Section 2 gives an extended version of the GRP transform. We extend the transform to allow arbitrary parameters. Our main emphasis is on the development of an efficient inverse transformation. For this, we apply the technique by Nong, Zhang, and Chan [5], [6] to our transform. Its details and other remarks on complexity issues will be given separately in Section 3.

2 GRP Transform with Arbitrary Parameters

2.1 Preliminaries

We consider lossless compression of strings over an ordered alphabet $\mathcal{A}$ of a finite size $|\mathcal{A}|$. Elements of the alphabet are called *symbols*. The alphabetically largest symbol in the alphabet, denoted by \$, is a *sentinel*, which appears exactly once at the end of a data string. We represent a string of length n by

$$x[1..n] = x_1 x_2 \cdots x_{n-1}\$, \tag{1}$$

where the ith symbol is denoted by $x[i]$, i.e., $x[i] = x_i$ for $1 \leq i \leq n-1$ and $x[n] = \$$. Similarly, a two-dimensional $m \times n$ matrix M of symbols is denoted by $M[1..m][1..n]$. The purpose of introducing sentinels is twofold. While the first one is the same as that adopted in the explanation of the BWT, the second one is specific to the present paper. We append some extra sentinels to the input string during the transformation in order to make its length an integer multiple of a parameter.

The GRP transform, which is an extension of our previous one [3], converts an input string (1) into another string $y[1..n] \in \mathcal{A}^n$. The transform has two parameters: the *block length* ℓ and the (*context*) *order* d, where ℓ is an integer in $1 \le \ell \le n$ and d is an integer in $0 \le d \le n$. The values of these parameters and the string length n are shared by the forward and inverse transformations so that the original string (1) can be uniquely recovered from the transformed string $y[1..n]$.

In the forward transformation, the input string (1) is divided into blocks of the same length ℓ. We call an integer $b = \lceil n/\ell \rceil$ the *number of blocks* of the string. We first consider a string

$$x'[1..b\ell] = x_1 x_2 \cdots x_{n-1} \$ \cdots \$, \tag{2}$$

which is a concatenation of $x[1..n]$ and extra $b\ell - n$ sentinels. Let $x'[0] = \$$. Then, the forward transformation begins with a $b \times b\ell$ matrix $M = M[1..b][1..b\ell]$ whose (i,j) element is initialized as

$$M[i][j] = x'[(i-1)\ell + j \bmod b\ell], \quad 1 \le i \le b,\ 1 \le j \le b\ell. \tag{3}$$

The leftmost d columns and the rightmost ℓ columns of M are called the *reference part* and the *output part*, respectively.

Example: For $n = 11$, $d = 4$, $\ell = 3$, consider the string:

$$x[1..11] = \texttt{bacacabaca\$}. \tag{4}$$

Then, $b = \lceil 11/3 \rceil = 4$, $b\ell = 12$, and $x'[1..12] = \texttt{bacacabaca\$\$}$. The initial configuration of M is given below. Note that the string $x'[1..b\ell]$ appears as the first row, and each row is followed by its ℓ-symbol left cyclic shift. Thus, the concatenation of the rows of the left ℓ symbols forms the string $x'[1..b\ell]$. If we concatenate the rows of any consecutive ℓ columns in an appropriate order, we can recover $x'[1..b\ell]$.

$$M = M[1..4][1..12] = \left[\begin{array}{cccccccccccc} \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ \\ \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} \\ \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} \\ \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} \end{array}\right].$$

$$\underbrace{\qquad\qquad}_{\substack{\text{reference part} \\ (d \text{ columns})}} \qquad\qquad \underbrace{\qquad\quad}_{\substack{\text{output part} \\ (\ell \text{ columns})}}$$

The forward and inverse transformations can be described in terms of operations on the matrix M. The most basic operation is the sorting of the row vectors. Sorting is performed in a stable manner by using the entire row or its part as a sorting key.

In our previous paper [3], we defined the GRP transform on a matrix consisting only of the reference and output parts because other elements have nothing to do with the transform. In fact, the matrix representation is not essential to the transform. We simply adopt the above representation so that we can intuitively understand the relation of the transform with the BWT and the ST.

2.2 Forward Transformation

The forward transformation comprises sorting of the row vectors of M and output of its column vectors.

```
1. /* Initialization */
   Convert the input string into a matrix $M = M[1..b][1..b\ell]$ using (3);
2. Use the symbols of the reference part (first $d$ columns) of $M$ as a sort key,
   and sort the rows of $M$ lexicographically and stably;
   /* The matrix $M$ is said to be in state "A" immediately after Step 2. */
3. for $j := b\ell$ downto $b\ell - \ell + 1$ do
   (a) Output the symbols of the $j$th column of the current $M$ according to:
           $y'[(b\ell - j)b + i] := M[i][j]$ for $1 \le i \le b$;
   (b) if $j = b\ell - \ell + 1$ then break;
   (c) Sort the row vectors of $M$ in a stable manner by the symbols of the $j$th
       column;
   end for
```

When there are more than one sentinels in $x'[1..b\ell]$, the second and other succeeding sentinels are outputted from the last row. Therefore, we do not have to include these sentinels except for the first one in the transformed string. In this case, the number of symbols outputted from the forward transformation is equal to n. We represent the output of the corresponding transformation by $y[1..n] = y_1 y_2 \ldots y_n$. The output drawn directly from the above procedure has been denoted by $y'[1..b\ell]$. Both $y[1..n]$ and $y'[1..b\ell]$ are easily converted to one another by sharing the parameters d, ℓ, and n.

Example: For the string in (4) and $d = 4$, $\ell = 3$, we have

$$y'[1..12] = \texttt{cc\$acaa\$bbaa}, \tag{5}$$

$$y[1..11] = \texttt{cc\$acaabbaa}. \tag{6}$$

2.3 Relation with Existing Transforms

Before proceeding to a description of the inverse transformation, we reveal close relations between the GRP transform and other established transforms.

First, consider the case of $\ell = 1$ and $d = n$. In this case, it is easy to see that $y[1..n]$ $(= y'[1..b\ell])$ is exactly the same as one obtained when we apply the BWT to $x[1..n]$. In this sense, the GRP transform can be regarded as a proper extension of the BWT. Similarly, we can consider the case where we limit the order d to an integer $k < n$ with $\ell = 1$. This is known as the k-order variant of the BWT, or sometimes called the Sort Transform (ST) [5],[6],[7].

Now, consider an application of the ST to a string over $\mathcal{A}^\ell$. For example, suppose that we have a string `CACB` over $\mathcal{A}^3$, where `A` = `aca`, `B` = `a$$`, and

$\texttt{C} = \texttt{bac}$. The ST with any $k \geq 1$ transforms this string into CCBA. If we apply the GRP transform with $d = 1$ and $\ell = 1$ to the same string CACB, we have

$$M = \begin{bmatrix} \texttt{C} & \texttt{A} & \texttt{C} & \texttt{B} \\ \texttt{A} & \texttt{C} & \texttt{B} & \texttt{C} \\ \texttt{C} & \texttt{B} & \texttt{C} & \texttt{A} \\ \texttt{B} & \texttt{C} & \texttt{A} & \texttt{C} \end{bmatrix} \longrightarrow M_{\mathrm{A}} = \begin{bmatrix} \texttt{A} & \texttt{C} & \texttt{B} & \texttt{C} \\ \texttt{B} & \texttt{C} & \texttt{A} & \texttt{C} \\ \texttt{C} & \texttt{A} & \texttt{C} & \texttt{B} \\ \texttt{C} & \texttt{B} & \texttt{C} & \texttt{A} \end{bmatrix} = \begin{bmatrix} \texttt{a c a b a c a \$ \$ b a c} \\ \texttt{a \$ \$ b a c a c a b a c} \\ \texttt{b a c a c a b a c a \$ \$} \\ \texttt{b a c a \$ \$ b a c a c a} \end{bmatrix},$$

where M_{A} represents the matrix M in state A. With a simple observation, we can see that, in the case of $d = kl$, the application of the k-order ST to an $|\mathcal{A}^{\ell}|$-ary string is essentially equivalent to obtaining the matrix M in state A in the GRP transform with parameters ℓ and d for the corresponding $|\mathcal{A}|$-ary string. The main difference between the k-order ST for an $|\mathcal{A}^{\ell}|$-ary string and the GRP transform for the equivalent $|\mathcal{A}|$-ary string lies in Step 3 of the forward transformation of the GRP transform, in which $|\mathcal{A}|$-ary symbols are outputted symbolwise and sorted repeatedly according to their right neighbor columns.

In the sequel, we represent a matrix M in state A by M_{A}, which plays a similar role to a sorted matrix in the BWT and the ST.

2.4 Inverse Transformation

The inverse transformation of the GRP transform consists mainly of two parts: reconstruction of the output part of M_{A} and restoring the original string.

Let $L_\ell[1..b][1..\ell]$ denote the output part of M_{A}. That is, $L_\ell[i][j] = M_{\mathrm{A}}[i][(b-1)\ell+j]$ for $1 \leq i \leq b$ and $1 \leq j \leq \ell$. It can be reconstructed from the transformed string $y'[1..b\ell]$ in the following way.

```
/* Reconstruction of the output part of $M_{\mathrm{A}}$ */
1. Allocate a $b \times \ell$ matrix $L_\ell[1..b][1..\ell]$ of symbols, and set

       $L_\ell[i][1] := y'[i + (\ell - 1)b]$ for $1 \leq i \leq b$;

2. for $j := 2$ to $\ell$ do
   (a) Sort the symbols in $y'[1 + (\ell - j)b..b + (\ell - j)b]$ alphabetically, and put
       the result into $L_\ell[1..b][j]$;
   (b) Rearrange the rows of $L_\ell[1..b][1..j]$ in a stable manner so that its $j$th
       column corresponds to $y'[1 + (\ell - j)b..b + (\ell - j)b]$;
   end for
```

The validity of the above procedure was given in [3] as Lemma 1. The output part $L_\ell[1..b][1..\ell]$ plays a similar role to the last column of the sorted matrix of BWT. In BWT, the symbols in the last column are then sorted in order to obtain the first column of the same matrix.

In the GRP transform, on the other hand, we can use $L_\ell[1..b][1..\ell]$ to obtain the first d or ℓ columns of M_{A}, depending on the value of d. Actually, however, we do not recover explicitly the left columns of M_{A}. Instead, we keep only two mappings between the left and right ends of M_{A}. To do so, we perform stable sort on the set of row vectors of $L_\ell[1..b][1..\ell]$ in lexicographic order of their prefixes of length $\min\{d, \ell\}$. As a result of sorting, if $L_\ell[i][1..\ell]$ is the jth one of the sorted list of the row vectors, then define two column vectors: $P[1..b]$ and $Q[1..b]$ by

$$P[i] = j \text{ and } Q[j] = i.$$

When $d = 0$, they are defined by $P[i] = n$ for $i = 1$ and $P[i] = i - 1$ otherwise, and $Q[j] = 1$ for $j = n$ and $Q[j] = j + 1$ otherwise.

In order to continue our description of the inverse transform, we borrow some notions from [6], in which Nong, Zhang, and Chan developed a linear time implementation of the inverse ST. We first introduce a binary vector $D[1..b] \in \{0, 1\}^b$ such that

$$D[i] = \begin{cases} 0 & \text{for } i \geq 2 \text{ and } M_{\mathrm{A}}[i][1..d] = M_{\mathrm{A}}[i-1][1..d], \\ 1 & \text{for } i = 1 \text{ or } M_{\mathrm{A}}[i][1..d] \neq M_{\mathrm{A}}[i-1][1..d]. \end{cases} \tag{7}$$

Further, we introduce a counter vector $C_d[1..b]$ and an index vector $T_d[1..b]$. If $D[i] = 0$, then $C_d[i]$ is also defined to be zero. Otherwise, $C_d[i]$ stores the number of the same prefixes of the row vectors in M_{A} as $M_{\mathrm{A}}[i][1..d]$. The index vector $T_d[1..b]$ along with $C_d[1..b]$ is computed in the following way, provided that $D[1..b]$ is given. The computation of $D[1..b]$ will be deferred to the next section.

```
   /* Computation of $T_d$ and $C_d$ */
1. Set $C_d[1..b]$ to be a zero vector;
2. **for** $i := 1$ **to** $b$ **do**
   (a) **if** $D[i] = 1$ then set $j := i$;
   (b) Set $T_d[Q[i]] := j$;
   (c) Set $C_d[j] := C_d[j] + 1$;
   **end for**
```

We are now ready to summarize the inverse transformation. We can restore the original string $x'[1..b\ell]$ using the following procedure.

```
   /* Restoring the original string */
1. Set $j := (b-1)\ell + 1$;
2. Set $i :=$ such an index that $L_\ell[i][1..\ell]$ includes the sentinel $;
3. **while** $j > 0$ **do**
   (a) Set $x'[j..j+\ell-1] := L_\ell[i][1..\ell]$;
   (b) Set $i := T_d[i]$;
   (c) Set $C_d[i] := C_d[i] - 1$;
   (d) Set $i := i + C_d[i]$;
   (e) Set $j := j - \ell$;
   **end while**
```

Example: First, the transformed string in (6) is converted to its equivalent form (5) by using $n = 11$ and $\ell = 3$. Then, $L_\ell[1..b][1..\ell]$ and other auxiliary quantities are obtained as shown in the left table. Finally, the original string $x'[1..12]$ is restored as shown right below.

i	$L_3[i][1..3]$	$Q[i]$	$D[i]$	$T_4[i]$	$C_4[i]$
1	`bac`	4	1	3	1
2	`bac`	3	1	3	1
3	`a$$`	1	1	2	2
4	`aca`	2	0	1	0

j	$L_3[i][1..3],\ i = 3$
10	$x'[10..12] =$ `a$$`, $i = 2$
7	$x'[7..9] =$ `bac`, $i = 4$
4	$x'[4..6] =$ `aca`, $i = 1$
1	$x'[1..3] =$ `bac`

3 Details of the Inverse Transformation

3.1 Computation of Vector *D*

The inverse transformation above is based on the framework of Nong and Zhang [5]. A key issue in the framework lies in the computation of D, which they called the *context switch* vector. The same authors gave a more efficient solution to the issue in their recent paper [6]. We here generalize their recent technique in order to apply it to the computation of the vector $D[1..b]$ in our case.

As shown in (7), the vector $D[1..b]$ is defined on the matrix in state A, which is not available when we are about to compute $D[1..b]$. However, the left d columns of $M_{\mathrm{A}}[1..d][1..b\ell]$ can be retrieved only by $L_\ell[1..b][1..\ell]$ and $Q[1..b]$ in the following way.

```
   /* Reconstruction of the $i$th row of the reference part of $M_{\mathrm{A}}$ */
1. Set $j := 1$ and $k := Q[i]$;
2. while $j \le d$ do
   (a) Set $M_{\mathrm{A}}[i][j..j+\ell-1] := L_\ell[k][1..\ell]$;   /* valid only up to $d$ columns */
   (b) Set $k := Q[k]$;  $j := j + \ell$;
   end while
```

When we wish to have a single value $D[i]$ for a specific i, we may perform the above procedure for $i-1$ and i, and compare the results symbol by symbol. However, in order to obtain the set $D[1..b]$ of those values more efficiently, we rather use a new quantity *height* and the notion of *cycles*. Thus, the above reconstruction procedure is no longer used in actual transformation, but should be remarked as a procedural representation of *Property 6* in [6] with being generalized to our transform. Property 6 in [6] says that a limited (constant) order context of the ST can be retrieved by the combination of mapping Q and symbols in the last column.

The vector $height[1..b]$ represents the lengths of the longest common prefixes (LCPs) between adjacent rows in M_{A}. Let $height[i]$ denote the length of the LCP between $M_{\mathrm{A}}[i-1][1..b\ell]$ and $M_{\mathrm{A}}[i][1..b\ell]$. Obviously, $\{D[i] = 0\}$ is equivalent to $\{height[i] \ge d\}$ for $2 \le i \le b$. The following theorem is a generalization of Theorem 1 in [6] to the GRP transform with parameters d and ℓ. The original theorem in [6] corresponds to the case of $\ell = 1$.

Theorem 1. *$height[Q[i]] \geq height[i] - \ell$ for $2 \leq i \leq b$.*

Both in the above procedure and in the theorem, the indexes to the row vectors are retrieved one after another by the use of mapping Q. Starting from an arbitrary integer i in $[1, b]$, we eventually return to the same integer after successive applications of Q. Thus, it is natural to define a set of indexes

$$\alpha(i) = \{i, Q[i], Q^2[i], \ldots, Q^{b-1}[i]\}, \tag{8}$$

where $Q^k[i]$ represents $Q[i]$ for $k = 1$ and $Q[Q^{k-1}[i]]$ for $k \geq 2$. We call $\alpha(i)$ a *cycle*, which we may regard as either a set or a linear list of indexes (integers) depending on the context. Obviously, any two cycles are disjoint. Although our definition of cycles is slightly different from that in [6], where a cycle is defined as a list of *symbols*, we can apply almost the same discussions as those in [6] to the computation of D in the inverse GRP transformation. The most characteristic difference arises when we apply Theorem 1 to the computation of *height*s along a cycle. A specific procedure follows.

```
/* Computation of the heights for the indexes in a cycle $\alpha(i)$ */
1. Set $j := i$ and $h := 0$;
2. **do**
   (a) **while** $h < d$ **do**
         **if** $j = 1$ or $Diff(j, h)$ **then** break **else** $h{+}{+}$;
       **end while**
   (b) Set $height[j] := h$;
   (c) Set $h := \max\{h - \ell,\ 0\}$;
   (d) Set $j := Q[j]$;
   **while** $j \neq i$
```

In the above procedure, $Diff(j, h)$ is a boolean predicate that indicates whether the hth symbols ($1 \leq h \leq d$) of the $j - 1$th and jth row vectors of M_{A} are different. If we can perform this comparison in $O(1)$ time, the time complexity of the above procedure becomes $O(d + \ell \cdot |\alpha(i)|)$ for the cardinality $|\alpha(i)|$ of cycle $\alpha(i)$. This comes from the fact that the total number of times we increment h in Step 2 (a) never exceeds the sum of d and the total number of decrements of h in Step 2 (c). Since the sum of the cardinalities $|\alpha(i)|$ of all the cycles is equal to $b = \lceil n/\ell \rceil$, the essential issues to be addressed when we wish to compute the vector D in linear time are the development of $O(1)$-implementation of $Diff(j, h)$ and the exclusion of d from the complexity of the above procedure for all cycles. Actually, these two issues are the main focus of Nong, Zhang, and Chan [6].

We can apply their method [6] almost as it is in order to compute $D[1..b]$ in time linear in the string length. First, to implement $Diff(j, h)$, note that the hth symbol of the jth row of M_{A} is given by

$$M_{\mathrm{A}}[j][h] = L_\ell[Q^{\lceil h/\ell \rceil}[j]][(h-1) \bmod \ell + 1] \quad \text{for } 1 \leq j \leq b,\ 1 \leq h \leq d. \tag{9}$$

This can be validated by the reconstruction procedure of the reference part of M_{A}, which was given in the beginning of this subsection. In (9), $Q^{\lceil h/\ell \rceil}[j]$ can be computed in a constant time by the use of a suitable data structure [6]. Thus, we can compute $\mathit{Diff}(j,h)$ in $O(1)$-time from $L_\ell[1..b][1..\ell]$.

As for the exclusion of d from the complexity $\sum_{\text{cycles}} O(d+\ell\,|\alpha(i)|)$, we can completely follow the technique in [6]. As a result, we can compute the vector $D[1..b]$ in $O(\ell b)=O(n)$ time without depending on the value of d.

3.2 Summary of Complexity Issues

We summarize the complexity of the inverse transformation.

The inverse transformation begins with the reconstruction of the output part $L_\ell[1..b][1..\ell]$ of M_{A}. Assuming that sorting of symbols can be performed in linear time using bucket sorting, we can reconstruct the output part in time linear in the string length n. Then, we proceed to the computation of the mappings $P[1..b]$ and $Q[1..b]$. These mappings are obtained by the sorting of the set of row vectors of $L_\ell[1..b][1..\ell]$. Since the number of the row vectors is b and the length of the key is at most ℓ symbols, the mappings are produced by radix sort in $O(b\ell)=O(n)$ time.

The computation of vectors $T_d[1..b]$ and $C_d[1..b]$ can be done obviously in time linear in b when we already have the vector D, which we can obtain in $O(n)$ time, as mentioned above. The last operation for restoring the original string is to simply copy the set of row vectors of $L_\ell[1..b][1..\ell]$ in a designated order. Thus, we can now have the following theorem.

Theorem 2. *The inverse transformation of the proposed GRP transform can restore the original string of length n in $O(n)$ time.*

Although we have not discussed the space complexity so far, it is obvious that the space requirement for the inverse transformation is also $O(n)$ because $L_\ell[1..b][1..\ell]$ requires $O(n)$ space while other auxiliary vectors P, Q, D, T_d, and C_d require only $O(b)$ space.

To conclude this subsection, we must give a brief comment on the complexity of the forward transformation. Its time complexity is obviously $O(b(d+\ell)) = O(bd+n)$ when we implement it by using simple radix sort. Although it is desirable to exclude the order d as in the case with the inverse transformation, $O(bd+n)$ is not so demanding in practice. If we use the same radix sort to implement the k-order ST, the time complexity of the k-order ST will be $O(kn)$. By choosing appropriate parameters in the GRP transform, we can make its time complexity of $O(bd+n)$ significantly smaller than $O(kn)$ of the k-order ST. When the GRP transform corresponds to the BWT, we can use various techniques developed for faster construction of a suffix array [1]. However, it does not seem plausible to be able to generalize those techniques to the GRP transform. On the other hand, the space requirement of the forward transformation is simply $O(n)$. An array for the input string with a permutation vector representing the row order of M will suffice to implement the matrix. We can access its elements via the relation (3).

4 Conclusion

We have proposed an extension of the GRP transform and its efficient implementation for the inverse transformation. The GRP transform is a proper generalization of the BWT and their typical variations. We can also say that, in an algorithmic viewpoint, the proposed inverse transformation is a generalization of the Nong–Zhang–Chan implementation of the inverse ST [6]. The generality of the GRP transform will result also in the generality of second-step encoders, which are used after the transform for actual compression. We can extend the Move-to-Front heuristics, an example of the second-step encoders incorporated into the block-sorting compression algorithm, so that it accommodates the characteristics of the output of the GRP transform. We will present its details with compression experiments on another occasion.

Acknowledgement

We thank Kazumasa Inagaki and Yoshihiro Tomizawa for their contribution to our previous work.

References

1. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, Heidelberg (2008)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Systems Research Center, Palo Alto (1994)
3. Inagaki, K., Tomizawa, Y., Yokoo, H.: Novel and generalized sort-based transform for lossless data compression. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 102–113. Springer, Heidelberg (2009)
4. Inagaki, K., Tomizawa, Y., Yokoo, H.: Data compression experiment with the GRP transform (in Japanese). In: 32nd Sympo. on Inform. Theory and its Applications, SITA 2009, Yamaguchi, Japan, pp. 330–335 (2009)
5. Nong, G., Zhang, S.: Efficient algorithms for the inverse sort transform. IEEE Trans. Computers 56(11), 1564–1574 (2007)
6. Nong, G., Zhang, S., Chan, W.H.: Computing inverse ST in linear complexity. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 178–190. Springer, Heidelberg (2008)
7. Schindler, M.: A fast block-sorting algorithm for lossless data compression. In: DCC 1997, Proc. Data Compression Conf, Snowbird, UT, p. 469 (1997)
8. Vo, B.D., Manku, G.S.: RadixZip: Linear time compression of token streams. In: Very Large Data Bases: Proc. 33rd Intern. Conf. on Very Large Data Bases, Vienna, pp. 1162–1172 (2007)