Electronics and Communications in Japan, Part 3, Vol. 82, No. 6, 1999
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J81-A, No. 3, March 1998, pp. 437–444

Notes on Block-Sorting Data Compression

Hidetoshi Yokoo

Department of Computer Science, Gunma University, Kiryu, Japan 376-8815

SUMMARY

The block-sorting data compression method of Burrows and Wheeler has received considerable attention in anticipation that it may be comparable, or even superior, to the Ziv–Lempel codes. This article discusses its characteristic points from the viewpoint of string algorithms. The block-sorting compression algorithm initially sorts all rotations of an input text lexicographically. This transformation is still reversible when we restrict the key length of sorting. We first focus on the combinatorial aspect of this reversibility, then show that the block-sorting algorithm has the capability of finding repetitions in an original string. We give a new description of the algorithm in terms of the Karp–Miller–Rosenberg (KMR) repetition finder in order to combine the above two aspects. © 1999 Scripta Technica, Electron Comm Jpn Pt 3, 82(6): 18–25, 1999

Key words: Data compression; text compression; universal codes; Burrows–Wheeler transform; KMR algorithm.

1. Introduction

We analyze the block-sorting data compression algorithm [1], which has received considerable attention in anticipation that it may outperform the Ziv–Lempel (LZ) codes in string algorithms.

The block-sorting data compression algorithm is an off-line method, while most conventional text compression methods are on-line adaptive methods. In spite of the off-line nature, it particularly attracts attention because of its simple and elegant operation and excellent practical compression performance [2–6]. Most previous studies, however, consist of experimental evaluations in which comparisons are made between its original version and some modifications, adopting possible alternatives to the sorting and encoding phases of the algorithm. On the other hand, the present author proposes the context-sorting compression method [7], which is, in a sense, an on-line version of the block-sorting method, and discusses limiting the key length of sorting to a fixed number of symbols, the asymptotic optimality of the proposed method, and its relationship to the LZ codes. In this article, we discuss two of these topics: a method for limiting the key length of sorting and the relation to the LZ codes.

In the next section, we summarize the block-sorting data compression algorithm. The algorithm begins with lexicographic sorting of all cyclic shifts of an input text called a block. This sometimes requires lexicographic comparisons of quite long strings, depending on the adopted sorting algorithm. In practice, however, such an exact comparison is not necessarily needed. In section 3, we generalize the algorithm by limiting the sort key to prefixes of bounded length and discuss a decoding method corresponding to this generalization. A similar extension to bounded key cases has been already proposed in Ref. 8. However, its decoding algorithm is not sufficiently refined. Here we give a clearer description of the reversibility of the generalized block-sorting algorithm. Section 4 is devoted to the relationship between the LZ codes and the block-sorting algorithm. The LZ codes, including many variations, are designed as on-line algorithms. On the other hand, the block-sorting method, which has an offline nature, may be regarded as an offline LZ scheme. In this paper, we design a concrete algorithm that embodies the idea of the offline

CCC1042-0967/99/060018-08

LZ scheme and show that the block-sorting algorithm has the capability of finding repetitions in a string that is necessarily associated with the LZ codes. Finally, in section 5, we combine the preceding discussions by presenting a new description of the block-sorting algorithm in terms of the KMR algorithm [9, 10].

2. Block-Sorting Data Compression

Consider the lossless compression of an input string over an ordered alphabet A of size α. Let $<$ denote the lexicographic order on A^* associated with the ordering relation on A. Here, A^* denotes the set of all strings over A, including the empty string λ.

Let

$$x[1, n] = x_1 x_2 \dots x_n \quad (x_i \in A, 1 \le i \le n) \tag{1}$$

be an input string of length n over A. We represent a substring of x as

$$x[i, j] = x_i x_{i+1} \dots x_j \tag{2}$$

and define $x[i, j] = \lambda$ for $i > j$. Starting from $U_1 = x[1, n]$, we generate all its cyclic shifts:

$$\begin{aligned} U_2 &= x_2 x_3 \dots x_n x_1 \\ U_3 &= x_3 \dots x_n x_1 x_2 \\ &\vdots \\ U_n &= x_n x_1 \dots x_{n-1} \end{aligned} \tag{3}$$

We represent a lexicographically sorted sequence of the n strings U_1 to U_n as

$$H_1 < H_2 < \cdots < H_n \tag{4}$$

Let I be an index $(1 \le I \le n)$ that satisfies $x[1, n] = H_I$. Furthermore, letting $H_j[k]$ be the kth symbol in H_j, we define

$$V_k[1, n] = H_1[k] H_2[k] \dots H_n[k] \tag{5}$$

for $1 \le k \le n$. In particular, we set

$$y[1, n] = y_1 y_2 \dots y_n = V_n[1, n] \tag{6}$$

The block-sorting compression algorithm converts the original string $x[1, n]$ into the $(y[1, n], I)$ pair, and encodes the pair in an appropriate way. The conversion that generates $(y[1, n], I)$ from $x[1, n]$ is sometimes called the Burrows–Wheeler (BW) transform [4].

There may be various ways of encoding the BW-transformed string $y[1, n]$, most of which are based on the idea of a symbol-ranking compressor [11]. We define the rank r_i of the ith symbol y_i in $y[1, n]$ in the following way. The rank r_1 of the first symbol is undefined. In $2 \le i \le n$, if the same symbol as y_i never appears in $y[1, i-1]$, then r_i is also undefined. Otherwise, the rank r_i is defined as the number of distinct symbols included in a set S_i that satisfies the following conditions:

$$\begin{aligned} &1 \le j < i \\ &y_j = y_i \\ &y_i \notin S_i \triangleq \{y_{j+1}, y_{j+2}, \dots, y_{i-1}\} \end{aligned} \tag{7}$$

For example, the string $x[1, 8] = bacacaba$ is converted into the following, where we adopt the alphabetic order as the ordering relation on A:

$$\begin{aligned} x[1, 8] &= b\ a\ c\ a\ c\ a\ b\ a \\ y[1, 8] &= c\ b\ c\ b\ a\ a\ a\ a \\ r[1, 8] &= -\ -\ 1\ 1\ -\ 0\ 0\ 0 \end{aligned} \tag{8}$$

In this example, the BW transform yields the index $I = 6$. We have also shown the sequence $r[1, 8]$ of the corresponding rank, in which we use the symbol – to represent an undefined rank. We may begin with an ordered list of all the symbols from A in order to avoid undefined ranks. Such ranks can be easily obtained by applying a self-organizing data structure, called the move-to-front (MTF) list [12, 13] to a BW-transformed string. We can encode a sequence $r[1, n]$ of ranks in such a way that we can uniquely recover $y[1, n]$ from the resulting codeword.

The BW transform is a reversible transformation. We can reconstruct $x[1, n]$ from a $(y[1, n], I)$ pair. To explain the reverse transformation, we add subscripts to indicate the position of each symbol in $y[1, n]$. We first sort the symbols in $y[1, n]$ using the alphabetic order of A in a stable manner. Stable sorting reserves the order of the same symbols. If we represent the sorted version of $y[1, n]$ by $z[1, n]$, we have

$$\begin{aligned} y[1, 8] &= c_1\ b_2\ c_3\ b_4\ a_5\ a_6\ a_7\ a_8 \\ z[1, 8] &= a_5\ a_6\ a_7\ a_8\ b_2\ b_4\ c_1\ c_3 \end{aligned}$$

Then, we write

$$\pi(j) = i$$

for the symbol y_i if it appears as the jth symbol z_j in $z[1, n]$ [6]. In the present example, we have $\pi(1) = 5$, $\pi(2) = 6$, $\pi(3) = 7$, and so on. We also write

$$z[1, n] = \pi y[1, n]$$

for

$$z_j = y_{\pi(j)} \quad (1 \le j \le n)$$

We call π the stable permutation that generates $z[1, n]$ from $y[1, n]$. Beginning with $w := \lambda$, $i := I$, we repeat

$$i := \pi(i)$$

$$w := w \cdot y_i$$

n times to recover the original string $x[1, n]$ in w. Here, the symbol • represents a string concatenation operation. The same result can also be obtained by the iteration $w := w \cdot z_i$; $i := \pi(i)$. The validity of this transform is discussed in the next section.

3. Parameterization with Context Length

An intuitive explanation of the compression power of the block-sorting compression algorithm may be that the BW-transformed string $y[1, n]$ is more "compressible" than the original string $x[1, n]$. When we lexicographically sort a set Eq. (3) of strings, an adjacent string pair may share a common prefix. That is, symbols that share a following context [5] gather in $y[1, n]$. Thus, the BW transform is a permutation operation that arranges symbols according to context similarity. Although the BW transform has no explicit limit on the context length, the lexicographic order of Eq. (3) on actual data is usually determined by several leading symbols. In addition, we cannot expect higher performance if we take a longer context into account. We should thus consider a modified block-sorting compression algorithm incorporating a sorting procedure for contexts of finite length. Such a modified algorithm is of practical interest because it reduces the time required for lexicographic sorting compared to the generation of fully sorted sequences of symbols. In this section, we show that there certainly exists such a modified algorithm.

First, we define the relation $<_M$ to be the lexicographic order of M-symbol prefixes. The case $M \ge n$ corresponds to the original BW transform. We use a stable sorting operation to sort Eq. (3) when $0 \le M < n$. Let $H_1 <_M H_2 <_M \ldots <_M H_n$ denote the stably sorted version of Eq. (3). Similar to the notation used in the previous section, we define

$$V_k[1, n] = H_1[k]H_2[k] \ldots H_n[k] \tag{9}$$

The output of the generalized BW transform is given by

$$y[1, n] = V_n[1, n] \tag{10}$$

If we alphabetically sort the symbols in $y[1, n] = V_n[1, n]$ in a stable manner, then the result agrees with $V_1[1, n]$. Let π_M denote the stable permutation that generates $V_1[1, n]$ from $V_n[1, n]$. Then, we can show the following result.

Theorem 1. *For* $1 \le i < \min\{M, n\}$, we have

$$V_{i+1}[1, n] = \pi_M V_i[1, n] \tag{11}$$

Proof. Let $H_j[0, M]$ denote the concatenation of $H_j[n]$ with $H_j[1, M]$, that is,

$$H_j[0, M] = H_j[n]H_j[1, M] \quad (1 \le j \le n)$$

We have a one-to-one correspondence between $\{H_j[0, M-1]\}_{j=1}^n$ and $\{H_j[1, M]\}_{j=1}^n$. Note that $\{H_j[1, M]\}_{j=1}^n$ is lexicographically sorted. First, consider the case where

$$H_j[0] = H_{j+1}[0] \tag{12}$$

for some $j (1 \le j \le n)$. For the index j in Eq. (12), there exists a unique i such that

$$j = \pi_M(i)$$

Assumption (12) can be combined with the relation $H_j[1, M] < H_{j+1}[1, M]$ to yield

$$H_j[0, M-1] < H_{j+1}[0, M-1]$$

Since the sorting operation is performed in a stable manner, we have

$$H_i[1, M] = H_j[0, M-1]$$
$$H_{i+1}[1, M] = H_{j+1}[0, M-1]$$

Thus, the equality

$$H_i[1, M] = H_{\pi_M(i)}[0, M-1] \tag{13}$$

is established whether relation (12) holds or not. This completes the proof of Eq. (11). □

We can use Theorem 1 to recover the ordered set of $H_i[1, M]$s from the BW transformed string $y[1, n]$. Instead of explaining the use of the theorem, we give an example that intuitively illustrates the theorem. We first show the result that is obtained by sorting the rotations of $x[1, 8]$ = *bacacaba* in $<_M$ order with $M = 3$ in Fig. 1(a). In this example, the original string $x[1, 8]$ is in the sixth row, which happens to gives the same index $I = 6$ as that in the previous section. If we concatenate the rightmost symbols in Fig. 1(a), we then have the transformed string

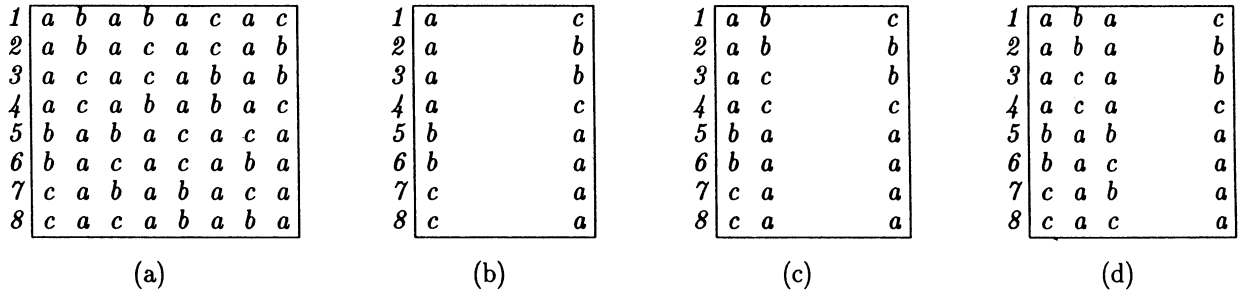

Fig. 1. A decoding example of prefixes, $M = 3$.

$$y[1, 8] = c\ b\ b\ c\ a\ a\ a\ a \tag{14}$$

This result is slightly different from the one in Eq. (8). Starting from this transformed string, we can recover the ordered set of $H_i[1, M]$s as follows. As shown in Fig. 1(b), we first put the elements of $y[1, 8]$ in the rightmost column, and then arrange the same elements in alphabetic order in the first column. We now know that the original string has adjacent symbol pairs (*digrams*), including a cyclic shift: *ca*, *ba*, *ba*, *ca*, *ab*, *ab*, *ac*, *ac*. If we sort these digrams in lexicographic order, we have the first and second columns shown in Fig. 1(c). This, in turn, gives all trigrams that are included in the original string. All these trigrams are placed in lexicographic order to form Fig. 1(d). Generally, beginning with $j := i$, the prefix $H_i[1, M]$ of length M in the ith row can be obtained by repeating

$$j := \pi_M(j); \quad output(y_j)$$

M times for the transformed string $y[1, n]$. The reverse transform given in the previous section is a special case of this procedure where we recover only the Ith row for $M = n$.

Although we have obtained an ordered set of prefixes of M symbols so far, we cannot continue using the same procedure. The rest of an input string, the postfix, must be recovered based on the stability of the sorting procedure. Consider again the example in Fig. 1. The first and second rows in (d) show that the original string has two occurrences of *aba*, which are preceded by *c* and *b*. In the original string $x[1, 8]$, the substring *aba* corresponding to the first row precedes that corresponding to the second row because they are sorted in a stable manner. Conversely, *aba* in the second row succeeds the same substring found in the first row. Now, observe the sixth (= Ith) row in Fig. 1(d), which shows that we have *a* immediately before *bac*. The symbol prior to that *a* is identical with a symbol that precedes *aba*. It should be the last of the symbols that occurs just before *aba*, since the Ith row is the original string $x[1, 8]$. We know from the second row that it should be *b*, which is inserted into Fig. 2(a). In Fig. 2(a), we have deleted the second row because we have already used it as a production rule. We repeat a similar procedure to produce matrices Fig. 2(b) to Fig. 2(d), in which we finally recover the original string.

As shown above, a modified version of the block-sorting algorithm is still reversible when we restrict the key length of sorting. The original version requires more time in encoding than in decoding. Although our algorithm generalized in this paper decreases the time required for sorting in encoding, it increases the complexity of decod-

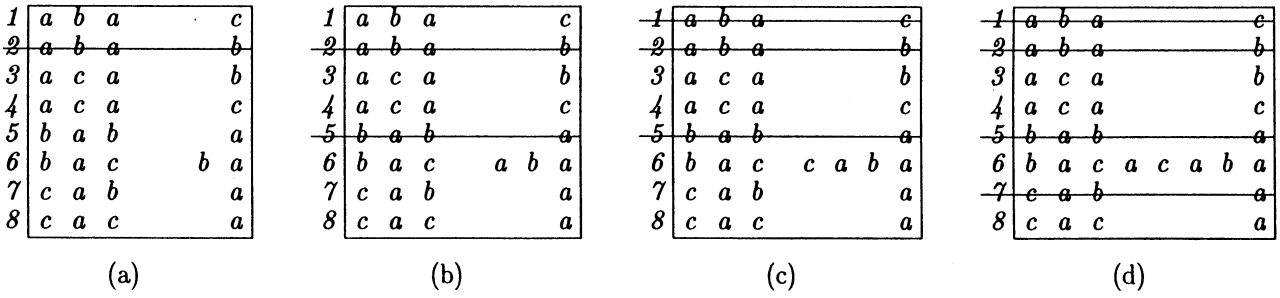

Fig. 2. A decoding example of postfixes, $M = 3$.

ing. In particular, the decoding method described in this section must compute and store all the prefixes $\{H_i[1, M]\}_{i=1}^{n}$. However, there are cases where we can use the original reverse BW transform even for bounded M, as described in section 5.

4. Finding Repetitions

As well as the usual lexicographic ordering, we can incorporate the reverse lexicographic ordering into the block-sorting algorithm. When we use the reverse lexicographic ordering, we take postfixes of length M into consideration, rather than prefixes. This makes it easier to understand the algorithm as a compression method for Mth-order Markov sources. However, when we consider all the cyclic shifts of the input string, we must concatenate its head and tail. Such an artificial operation prevents us from discussing methods for Mth-order Markov sources. On the other hand, we can avoid the cyclic shift if we introduce a special symbol that indicates the end of string [1]. From now on, we adopt such a convention in our discussion.

We introduce a special end marker \$, which is not included in A. The symbol \$ is assumed to precede lexicographically any symbol in A. Denote the concatenation of the input string with \$ by

$$x[1, n+1] = x_1 \ldots x_n\$ \tag{15}$$

We call this a tagged block, which is converted by the BW transform.

As an example, consider the input string $x[1, 12] =$ *obladioblada*. If we lexicographically sort all the cyclic shifts of its tagged block, then we have

$$\begin{aligned}
H_1 &= \$_{13} \\
H_2 &= a_{12}\$_{13} \\
H_3 &= a_{10}d_{11}a_{12}\$_{13} \\
H_4 &= a_4d_5i_6o_7b_8l_9a_{10}d_{11}a_{12}\$_{13} \\
&\vdots \\
H_{13} &= o_1b_2l_3a_4d_5i_6o_7b_8l_9a_{10}d_{11}a_{12}\$_{13}
\end{aligned}$$

In the above example, we add subscripts to indicate the position of each symbol in the tagged block of $x[1, n]$. We omit symbols after \$, which never affect the lexicographic order. In this manner, we can avoid string rotation. The last symbol of each row $H_i[1, n+1]$ is not explicitly represented. However, the last symbol can be easily known because it should occur immediately before $H_i[1]$ on the tagged block. In the above example, we can arrange the symbols $x_{12}, x_{11}, x_9, x_3, \ldots$ in this order to have the transformed string:

$$\begin{aligned}
y[1, 13] &= a\ d\ l\ l\ o\ o\ a\ a\ d\ b\ b\ i\ \$, \\
r[1, 13] &= -\ -\ -\ 0\ -\ 0\ 3\ 0\ 3\ -\ 0\ -\ -
\end{aligned}$$

where we have also shown the sequence of ranks. Note that, in the BW transform of the tagged block, the string ending with \$ should be the original string Eq. (15). Therefore, we can recover it from the transformed string $y[1, n+1]$ alone, without knowing the value of index I. If we use the position of \$ in $y[1, n+1]$ as an equivalent of the index I, we can apply an MTF coder to $y[1, n+1]$ after deleting \$ from it.

We know that the output of an MTF coder when applied to a BW-transformed string is dominated by the topmost rank (*rank* 0) [5] although it is not well observed in the above short example. Actually, several implementations realizing run length coding of the rank 0s have already been reported. The high frequency of rank 0 has been empirically observed but has not been sufficiently analyzed. Ranks are defined on the BW-transformed string. Here, however, we investigate the structure of runs of top-ranked symbols, not on BW-transformed strings, but on original strings. If we underline the symbols with rank 0 in the tagged block of the present example, then we have

$$x[1, n+1] = \underline{o}\,b\,\underline{l}\,\underline{a}\,dioblada\$$$

In this example, the underlined part *obla* reappears in the same string. In the following, we show that this is the case in general.

We define the following context of the ith symbol x_i the input string by $x[i+1, n+1]$. The BW transform of the tagged block is the lexicographic sorting of the set of contexts.

From now on, let $r_i = r$ denote the rank of x_i when the symbol x_i is transformed into y_k and the rank of y_k is r. Assume that two contexts $x[j+1, n+1]$ and $x[i+1, n+1]$ are adjacent to each other in the sorted set of contexts $H_1 < \ldots < H_{n+1}$ and that $x[j+1, n+1]$ immediately precedes $x[i+1, n+1]$. Then, the rank r_i of x_i is 0 iff $x_i = x_j$. If, for a string s of length k, we have

$$\begin{aligned}
&x[j+1, j+k] = x[i+1, i+k] = s \\
&x_{j+k+1} \neq x_{i+k+1}
\end{aligned}$$

then we call s the common context of the symbols x_j and x_i. The symbols x_{j+k+1} and x_{i+k+1} are said to form a discriminating symbol pair. If, again, we have $x_i = x_j$ and we have r distinct symbols between $x[i+1, n+1]$ and $x[j+1, n+1]$, which are not adjacent in the sorted set of contexts, then the rank of x_i is $r_i = r$ (see Fig. 3). In these two cases, if there is no such j, the rank r_i is undefined. Using these notions, we can show the following result.

Theorem 2. *If a substring $x[i+1, i+l]$ of length l has a run of successive rank 0s, i.e., $r_{i+1} = r_{i+2} = \ldots =$*

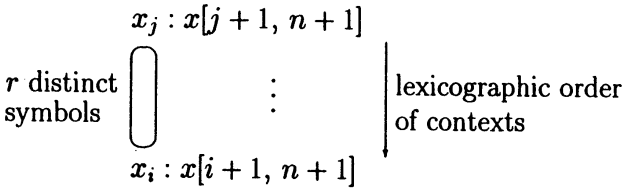

Fig. 3. The rank r_i of x_i is equal to the number r of distinct symbols between x_i and $x_j (= x_i)$.

$r_{i+l} = 0$, *and the rank* r_i *of the preceding symbol is greater than 0, then there exists another substring* $x[j+1, j+l]$ *that satisfies*

$$x[j+1, j+l] = x[i+1, i+l]$$

$$(0 \le j < i \ \text{or} \ i < j < n),$$

$$x_j \neq x_i$$

Proof. Let $k = i + l$. Since $r_k = r_{i+l} = 0$, there exists a immediate predecessor $x[k'+1, n+1]$ of $x[k+1, n+1]$ in the sorted set of contexts. It follows from $r_k = 0$ that $x_{k'} = x_k$. Let s denote the common context of $x_{k'}$ and x_k. That is, s is the longest common prefix of $x[k'+1, n+1]$ and $x[k+1, n+1]$. If the common context of $x_{k'-1}$ and x_{k-1} is longer than $x_k s$, it yields a contradiction. Thus, the common context of $x_{k'-1}$ and x_{k-1} should be $x_k s (= x_{k'} s)$. This leads to the fact that the context $x[k', n+1]$ immediately precedes $x[k, n+1]$ in the sorted set of contexts. Combining this fact with $r_{k-1} = 0$, we show that $x_{k'-1} = x_{k-1}$ in a similar manner. We can repeat this discussion as many times as the length l to prove $x[j+1, j+l] = x[i+1, i+l]$, where $j = k' - l$. The inequality $x_j \neq x_i$ is straightforward from the assumptions. □

Theorem 2 means that if there exists a run of rank 0s in the input string, then its corresponding substring reappears somewhere else in the string $x[1, n]$. The sign of $j - i$, that is, whether the repetition occurs on the right or left of the substring in question, depends on the discriminating symbol pair. If we represent the substring $x[i+1, i+l]$ in Theorem 2 by the $(j - i, l)$ pair, then we can define an internal macro encoding scheme [14]. The string $x[j+1, j+l]$ in Theorem 2 is a macro body [14] of the $(j - i, l)$ pointer. Although the coding scheme is recursive in the sense that a macro body may include pointers, the input string can be uniquely recovered by repeating substitutions of macro bodies for pointers. In the block-sorting compression, the encoding of runs of successive rank 0s corresponds to the encoding of repetitive substrings on the original string. However, the correspondence is not straightforward in actual encoding because the order of symbols is not preserved between the original and the transformed strings. In addition, Theorem 2 explicitly states the beginning of a repetition, but says nothing about its ending. That is, Theorem 2 never asserts $x_{j+l+1} \neq x_{i+l+1}$ for $r_{i+l+1} > 0$. It does not imply that any repetition can be detected as a substring starting with a rank-0 symbol. We see both a strong similarity and a large difference between the block-sorting compression and the offline macro scheme.

5. Generalization via the KMR Algorithm

The previous two sections dealt with two topics independently. In those sections, we considered nothing about the data structure problem for actual implementation. Neither did we discuss implementation issues required for any practical compression system. However, if we glimpse such an approach, we know that the discussions in those sections are strongly related to each other. In this section, we point out as an introduction to our approach that the discussions in the previous sections can interact via the KMR algorithm [9, 10], which is a tool for finding repetitions in a string.

As mentioned at the end of section 3, even in the method of restricting the key length of sorting to M symbols, the result of sorting for sufficiently large M is the same as that using the original BW transform. We can apply the original decoding algorithm to such a result. This situation occurs when the value of M is greater than the length of the longest repetition in the input string. The following method performs in parallel both the determination of such a value of M and the application of the BW transform with bounded M (and finally the original BW transform).

In this section, we again consider the BW transform of tagged blocks. Furthermore, we assume for convenience that we have an arbitrary number of \$s at the end of a tagged block. We assign a new quantity $NUM_t[i]$ to every position i of the original segment $x[1, n]$ of a tagged block $(t = 1, 2, \ldots; i = 1, \ldots, n)$. Define

$$NUM_t[i] = k \quad (k = 1, 2, \ldots) \tag{16}$$

if, when we sort lexicographically the set of all distinct strings of length t included in the tagged block, the substring $x[i, i+t-1]$ becomes the kth string. For instance, the example $x[1, 12] =$ *obladioblada* given in the previous section has the following sequence of $NUM_1[i]$:

$$NUM_1 = [6, 2, 5, 1, 3, 4, 6, 2, 5, 1, 3, 1]$$

The main point of the KMR algorithm is to obtain NUM_{2t} from NUM_t by lexicographically sorting the set of $NUM_t[i]$ and $NUM_t[i+t]$ pairs. If we continue with the above example, we have:

$$NUM_2 = [8, 3, 7, 2, 5, 6, 8, 3, 7, 2, 4, 1],$$
$$NUM_4 = [10, 4, 9, 3, 6, 7, 10, 4, 8, 2, 5, 1],$$
$$NUM_8 = [12, 5, 10, 3, 7, 8, 11, 4, 9, 2, 6, 1]$$

Obviously, by definition, if $NUM_t[i] = NUM_t[j]$ then $x[i, i + t - 1] = x[j, j + t - 1]$. For example, the equality $NUM_4[1] = NUM_4[7] = 10$ in the present example corresponds to $x[1, 4] = x[7, 10] = obla$. When the elements in NUM_t become all distinct, the BW transform $y[1, n + 1]$ of the tagged block is

$$y_1 = x_n \tag{17}$$

and, for Eq. (16) ($k = 1, 2, \ldots, n$),

$$y_{k+1} = \begin{cases} \$ & (i = 1) \\ x_{i-1} & (2 \leqq i \leqq n) \end{cases} \tag{18}$$

For t satisfying the condition, the fact that the rank of x_i is equal to 0 can be represented in the following. There exists j such that

$$NUM_t[j + 1] = NUM_t[i + 1] - 1$$

and $x_i = x_j$ holds for that j. We can use this relation to derive Theorem 2 from the previous section. It follows from the above discussion that the elements in NUM_t become all distinct when the value of t exceeds the length of the longest repetition in $x[1, n]$ (we have 5 for *oblad* in the above example). If we denote the length by L, we can perform the BW transform in $\log L + 1$ iterations, since the KMR algorithm doubles the value of t in each iteration. During the iterations, we can also convert NUM_t to the result corresponding to the BW transform for t-symbol prefixes.

We can easily extend the generation of NUM_t to any natural number t, though we have shown the KMR algorithm only for t to be 2s power.

6. Conclusions

Although excellent compression performance has been empirically observed in the bock-sorting data compression algorithm, most existing explanations for the performance are intuitive and qualitative. We have not discussed the compression performance in this paper, but we have presented more fundamental results regarding the generalization of the algorithm to cases of bounded key length and its capability of finding repetitions in a string. We have also shown that we not only can discuss these issues independently, but can relate one to the other via the KMR algorithm. In a theoretical analysis of the compression performance, we may either adopt the framework of Markov approximation of an information source or begin with the similarity of the block-sorting to representative universal codes of Ziv and Lempel. For example, if we regard the output of the KMR algorithm as a stochastic process, we can use Kac's lemma, a strong tool for analyzing a universal code. This article reveals constructive aspects of the basis for the analysis.

REFERENCES

1. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. SRC Research Report 1994;124.
2. Park JW, Imai H. A study on block-sorting data compression. Technical Report of IEICE 1995; 94:43–48.
3. Ei T, Inabeppu K, Uehara S. Note on data compression with block sorting. Technical Report of IEICE 1996;95:43–48.
4. Nelson M. Data compression with the Burrows–Wheeler Transform. Dr. Dobb's Journal, Sept. 1996, also available at: http://web2.airmail.net/markn/articles/bwt/bwt.htm.
5. Fenwick PM. Block-sorting text compression—final report. Technical Report 130, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1996. Anonymous ftp: ftp.cs.auckland.ac.nz, File: /out/peter-f/TechRep130.ps.
6. Arnavut Z, Magliveras SS. Block sorting and compression. DCC'97, In Storer JA, Cohn M, editors. Proc. Data Compression Conf. Snowbird. IEEE Computer Society Press; 1997. p 181–190.
7. Yokoo H. Data compression using a sort-based context similarity measure. The Computer Journal 1997;40:94–102.
8. Schindler M. A fast block-sorting algorithm for lossless data compression. In Storer A, Cohn M, editors. DCC'97, Proc. Data Compression Conf. Snowbird. IEEE Computer Society Press; 1997. p. 469. Extended version is available at: http://eiunix.tuwien.ac.at/~michael/papers/dcc97eab.ps.
9. Karp RM, Miller RE, Rosenberg AL. Rapid identification of repeated patterns in strings, trees and arrays. Proc 4th ACM Symposium on Theory of Computing 1972;125–136.
10. Crochemore M, Rytter W. Text Algorithms. New York: Oxford University Press; 1994.
11. Fenwick PM. Symbol ranking text compression. Technical Report 132, Department of Computer Science, University of Auckland, Auckland, New Zea-

land, 1996. Anonymous ftp: ftp.cs.auckland.ac.nz, File: /out/peter-f/TechRep132.ps.
12. Bentley JL, Sleator DD, Tarjan RE, Wei VK. A locally adaptive data compression scheme. Comm ACM 1986;29:320–330.
13. Elias P. Interval and recency rank source coding: Two on-line adaptive variable-length schemes. IEEE Trans Inf Theory 1987;IT-33:3–10.
14. Storer JA, Szymanski TG. Data compression via textual substitution. J ACM 1982;29:928–951.

AUTHOR

Hidetoshi Yokoo received his B.E. in instrumentation physics, his M.E. in information engineering, and his D.Eng. in mathematical engineering from the University of Tokyo, in 1978, 1980, and 1987, respectively. From 1980 to 1989 he was a research associate in the Department of Electrical Engineering, Yamagata University, Japan. In 1989, he joined Gunma University, Kiryu, Japan, where he is currently a professor in the Department of Computer Science. His research interests include data compression and its applications to computer science.