アルゴリズム 76－2
（2001．1．19）

ブロックソート圧縮データの検索法

外村 元伸†

†（株）日立製作所　中央研究所
〒185-8601　東京都国分寺市東恋ヶ窪 1-280
TEL (042)323-1111, E-mail:tonomura@crl.hitachi.co.jp

概要　BurrowsとWheelerによって1994年に報告された(元は1983年にWheelerによって提案された)ブロックソート圧縮アルゴリズムは，その圧縮率が，よく知られているLZ圧縮アルゴリズムに匹敵することから圧縮アルゴリズムの理論家のあいだで注目されている．本論文では，ブロックソート圧縮データに対してパターン検索する方法を提案する．そして，ブロックソート圧縮アルゴリズムはパターン検索しやすい効率的な圧縮法であることを指摘する．テキスト全体が巡回シフトの配列によって行が辞書式に順序付けられており，テキストの中で複数箇所に散在する検索パターンは，配列の行の先頭から現れ，連続する行に固まっている．検索パターンと効率的に照合するために，テキスト中で出現頻度の少ない検索パターン文字の位置から始めて一気に絞り込み，そのまわりに拡大する処理方式を提案する．そして，元のブロックソート圧縮アルゴリズムを修正して，配列の最後の列の文字列を圧縮符号化するかわりに，整列させた連続する文字の出現開始位置をポイントし，整列前の位置との対応付けを直接求められる圧縮符号化を提案する．

キーワード　圧縮，検索，ブロックソート圧縮，圧縮検索

Searching for Block Sorted Compression Data

Motonobu TONOMURA†

†Hitachi Central Research Laboratory, Hitachi, Ltd.
1-280 Higashi Koigakubo, Kokubunji-shi, Tokyo 185-8601, Japan
TEL (042)323-1111, E-mail:tonomura@crl.hitachi.co.jp

Abstract　The block sorting compression algorithm of Burrows and Wheeler (1994; originally proposed by Wheeler in 1983) has a compression ratio that is comparable to that of the well-known LZ compression algorithm. I have developed a pattern searching method for block sorted compression data. I show that the block sorting compression is especially effective in searching patterns. Since the whole text is sorted in lexicographic order with an array of the cyclic shifts, a searching pattern locates in several places in the text appears beginning from the first column and in contiguous rows of the array. The algorithm starts searching around characters that occur infrequently in the text. Furthermore, I describe a modified block sorting compression method that can directly obtain the correspondence of the previous and current sorting position, pointing to the start position of the sorting characters instead of the compressed coding of the last column of the array.

key words　compression, search, block sorting compression, compression search

1. Introduction

Data compression/decompression techniques efficiently store or transfer information, two types of LZ compression algorithms, created by Ziv and Lempel in 1977[23] and 1978,[20),24)] are the most widely used.[19] High-speed data searching methods,[11),13),16)~18)] on the other hand, require redundant information to search data. Thus, the methods of data compression and searching have developed in different directions, and little research has been done from the point of view of developing algorithms suitable to both data compression and search tasks. However, data compression and searching are closely connected because the processing of registrations and references into a compressed code dictionary is necessary to compress data.

Large quantities of data are normally compressed before storage. A search algorithm operating on this data must ideally be able to pick out particular references from a sea of compressed data. To search all the compressed data by decompressing and decoding is very wasteful. For that reason, effort has been made to find a method that searches within compressed codes. Amir[1),2)] proposed searching the compressed file directly and estimated the theoretical computational complexity. In practice, however, in the case of matching search patterns to compressed codes, the matching may extend in front of and behind the contents of the compressed code, which means the compressed codes of the search patterns cannot be uniquely determined.[14)]

While it is possible to look for an effective method that searches compressed codes,[6),9),10),12)] we may expect the same effect if compression is as fast as searching. The block sorting compression method, in particular, has a compression ratio equal to that of the LZ compression method, although how one achieves such a compression ratio is of theoretical concern.[5),7),21),22)] After some investigation, I found no reports related to searching methods for block sorted compression data.[1]

The block sorting compression method was the brainchild of D.J. Wheeler. He developed it in 1983 while he was working at AT&T Bell Laboratories, and M. Burrows followed this development with his own contribution in 1994.[4)] The method forms cyclic shifts (rotations) using all the text data, and all the cyclic shifts are sorted in lexicographical order and formed into an array. Encoding uses the last column of the array. Since the characters of a column are lined up in lexicographical order, compression coding them is simple. From the viewpoint of pattern searching, since the whole text is in lexicographical order with the array of cyclic shifts, in the case that a searching pattern locates in several places in the text, the searching pattern appears beginning from the first column of the array and in contiguous rows of the array. The block sorting compression method might therefore serve as an effective compression and search method.

[1] I knew the reports pointed out by K. Sadakane quite recently.[8),15)]

2. Block sorting data compression

Consider a text x of length n for which $x[1, n] = x_1 x_2 \cdots x_n$, where a character x_i ($1 \le i \le n$, i: position number) is an element of set A of the alphabet which has the lexicographic order "<" ($x_i \in A$). The subset characters of x can be expressed as $x[i, j] = x_i x_{i+1} \cdots x_j$ and $x[i] = x_i$. Then, the cyclic shifts of the text x are as $X_1 = x_1 x_2 \cdots x_n$, $X_2 = x_2 x_3 \cdots x_n x_1$, $\cdots$, $X_{n-1} = x_{n-1} x_n \cdots x_{n-2}$, $X_n = x_n x_1 \cdots x_{n-2}$. Sorting them in lexicographic order "<", we get $Y_1 < Y_2 < \cdots < Y_n$. Now, we select the last column $y[1, n] = Y_1[n] Y_2[n] \cdots Y[n]$ in a column of the array. This is called the Burrows-Wheeler transformation (BWT). Arranging y[i]'s in lexicographic order "<", we get z[1, n], namely, by a permutation π[3)]

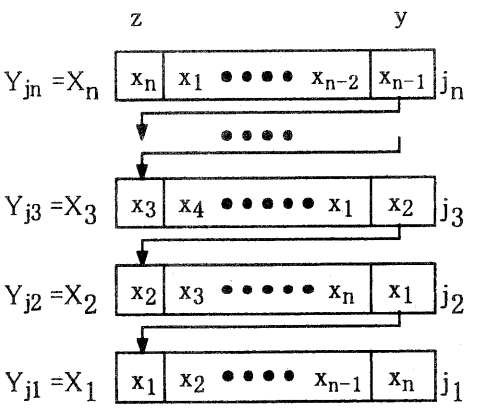

Fig. 1 Relation between sorting position numbers and permutation.

$$\pi = \begin{pmatrix} y[1] & y[2] & \cdots & y[n] \\ z[1] & z[2] & \cdots & z[n] \end{pmatrix}.$$

As shown in Fig. 1, $z[j_1] = x_1$ when $Y_{j1} = X_1 = x_1x_2\bullet\bullet\bullet x_n$. Next, since the cyclic shifts of X_1 are $X_2 = x_2x_3\bullet\bullet\bullet x_nx_1$, $z[j_2] = x_2$ such j_2 that $y[j_2] = x_1$. Similarly, $z[j_n] = x_n$ since the cyclic shifts of X_{n-1} are $X_n = x_nx_1\bullet\bullet\bullet x_{n-1}$. The reason that the last column, the $Y_1[n]Y_2[n]\bullet\bullet\bullet Y_n[n]$ characters of a column y[1, n], was selected is that the last column is permutated into the first column $Y_1[1]Y_2[1]\bullet\bullet\bullet Y_n[1]$ by the cyclic shifts. Thus, sorting and arranging characters of the last column in lexicographic order forms the first column. Then, since we got a pair of (current sorting position number, previous sorting position number) by the permutation, the code can be decoded into the original text by using the following relations if the previous sorting position number of the original text is known in advance.

The example of the block sorting compression method is shown in Fig. 2 (in outline) and in Fig. 3 (in detail) using a text 'cabccabcccabbcabccabcaccabbcaaab'. An array X[32, 32] of 32 cyclic shifts is generated from the

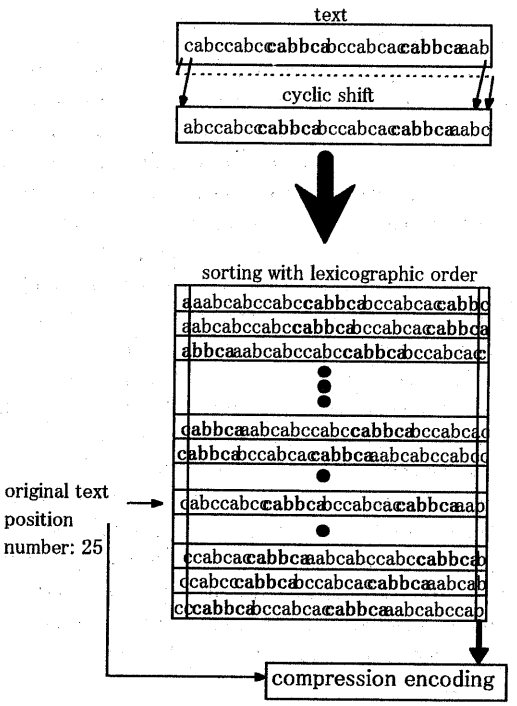

Fig. 2 Outline of block sorting compression method.

text 'cabccabcccabbcabccabcaccabbcaaab'

current sorting position nimber		previous sorting position nimber
01	aaabcabccabcccabbcabccabcaccabbc	02
02	aabcabccabcccabbcabccabcaccabbca	05
03	abbcaaabcabccabcccabbcabccabcacc	11
04	abbcabccabcaccabbcaaabcabccabccc	12
05	abcabccabcccabbcabccabcaccabbcaa	15
06	abcaccabbcaaabcabccabcccabbcabcc	16
07	abccabcaccabbcaaabcabccabcccabbc	17
08	abccabcccabbcabccabcaccabbcaaabc	18
09	abcccabbcabccabcaccabbcaaabcabcc	19
10	accabbcaaabcabccabcccabbcabccabc	28
11	bbcaaabcabccabcccabbcabccabcacca	13
12	bbcabccabcaccabbcaaabcabccabccca	14
13	bcaaabcabccabcccabbcabccabcaccab	20
14	bcabccabcaccabbcaaabcabccabcccab	24
15	bcabccabcccabbcabccabcaccabbcaaa	25
16	bcaccabbcaaabcabccabcccabbcabcca	27
17	bcccabcaccabbcaaabcabccabcccabbca	30
18	bcccabcccabbcabccabcaccabbcaaabca	31
19	bcccabbcabccabcaccabbcaaabcabcca	32
20	caaabcabccabcccabbcabccabcaccabb	01
21	cabbcaaabcabccabcccabbcabccabcac	03
22	cabbcabccabcaccabbcaaabcabccabcc	04
23	cabcaccabbcaaabcabccabcccabbcabc	06
24	cabccabcaccabbcaaabcabccabcccabb	07
25	cabccabcccabbcabccabcaccabbcaaab	08
26	cabcccabbcabccabcaccabbcaaabcabc	09
27	cacccabbcaaabcabccabcccabbcabccab	10
28	ccabbcaaabcabccabcccabbcabccabca	21
29	ccabbcabccabcaccabbcaaabcabcccabc	22
30	ccabcaccabbcaaabcabccabcccabbcab	23
31	ccabcccabbcabccabcaccabbcaaabcab	26
32	cccabbcabccabcaccabbcaaabcabcccab	29

Fig. 3 Example of block sorting data compression.

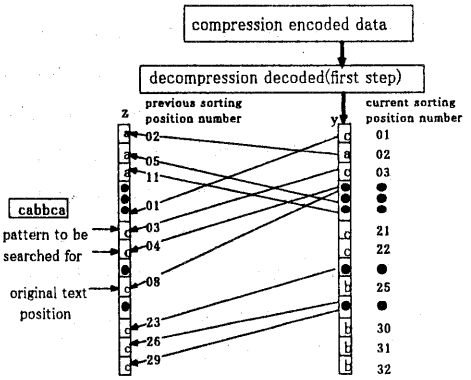

Fig. 4 Sorting of the last column and corresponding to the pair (current sorting position number, previous sorting position number)

text of length 32, sorting in lexicographic order such as 'a' < 'b' < 'c', and what is given a current sorting position number from 01 to 32. The original text is located in current sorting position number 25. From $Y_1[32]$ = 'c', $Y_2[32]$ = 'a', $Y_3[32]$ = 'c', •••, we get y[1, 32] = 'cacc•••cbbb'. We can increase the compression ratio, if we compress and encode a text using the run-length of characters since the line-up of characters of y[1, 32] is consecutive in the sense of 'a', 'b', 'c'. Thus, instead of a compression coding of the original text, the block sorting compression method is a coding y[1, 32] including the current sorting position number 25 of the original text. Since we can consider several methods in this compression and coding, we simply assume that y[1, 32] and the current sorting position number 25 can be compress and coded by a suitable one of several methods. Also, assuming that the compression code is decompressed and decoded up to this stage (the first stage of the compression and decoding), the method restoring to the original text (the second stage of the decoding) is as follows. First, sorting y[1, 32] in lexicographic order, we get z[1, 10] = 'aaa•••aaa', z[11, 19] = 'bbb•••bbb', and z[20, 32] = 'ccc•••ccc'. Then, since z[1] is arranged from y[2], z[2] is y[5], z[3] is y[11], •••, namely, is permutated, seeking for these corresponding relations, we store pairs of (current sorting position number, previous sorting position number) (see Fig. 4). As shown in Fig. 5, we can immediately see that z[25] = 'c' because we know that the current sorting position number of the original text is 25. Since the previous position number of the current sorting position number 25 is 08, we can restore up to 'ca', getting z[8] = 'a'. Then, since the previous position number of the current sorting position number 08 is 18, we can restore up to 'cab' getting z[18] = 'b'. Similarly, following pairs of (current sorting position number, previous sorting position number), we can restore the original text.

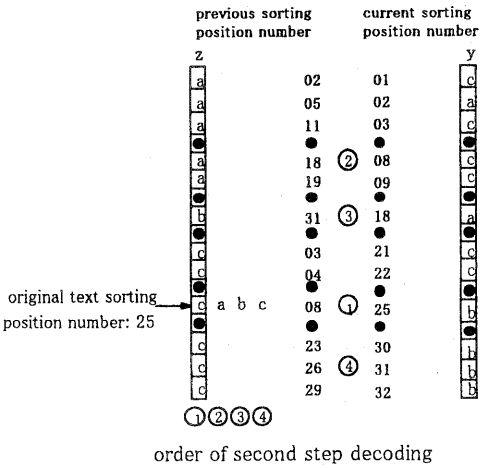

Fig. 5 Restoring the original text.

3. Pattern Searching

The block sorting compression algorithm for searching patterns requires the current sorting position number of the original text in order to decode the block sorting compression data into the original text. If we want to decode directly from any place in the text, we should know the current sorting position number of the position that appears in the head of the row of the array. Therefore, if necessary, we should also encode this sorting position number. But, since the searching patterns must be arbitrarily given, we cannot add them to the coding in advance. Giving only the searching patterns, we consider a case that matches them with the block sorting compression and coding data. However, since the first character of the searching pattern p[1] matches one of the first column characters of the array, we should specify the first position that the character appears in consecutive by sorting. It is easy to refer since the first position that the specified character appears contiguous rows of the array by sorting, and it is easy for us to know the frequency of each character.

In the text used in the example of Fig. 3, the search pattern of 'cabbca', appears in two places. Then, in the current sorting position numbers of

consecutive rows 21 and 22, we see that the pattern appears from at the first column position of the array.

p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
c	**a**	**b**	**b**	**c**	**a**
20	01				
21	**03**	**11**	**13**	**20**	**01**
22	**04**	**12**	**14**	**24**	**07**
23	06	16			
24	07	17			
25	08	18			
26	09	19			
27	10				
28					
29					
30					
31					
32					

Fig. 6 Matching the searching pattern p.

As shown in Fig. 6, since the first character of the pattern p is p[1] = 'c', the character c matches with the character of the first column of current sorting position numbers 20 to 32; c = $Y_{20}[1]$, •••, $Y_{32}[1]$. The current sorting position number that matches with the character c can be obtained from z[20, 32] = 'ccc•••ccc' when sorting the first column of the array. Since the first and second characters of the pattern p are p[1, 2] = 'ca', we seek for the pairs of (current position number, previous position number) matching this, (20, 01), (21, 03), (22, 04), (23, 06), (24, 07), (25, 08), (26, 09), (27, 10). Since the pair after (28, **) does not match with p[2] = 'a', it is removed from the candidates, Matching the pairs of (current position number, previous position number) with the second and third character p[2, 3] ='ab' of pattern p, we then get pairs of (03, 11), (04, 12), (06,16), (07, 17), (08, 18), (09, 19). Matching the third and fourth characters p[3, 4] ='bb' of pattern p are pairs of (11, 13) and (12, 14). Similarly, tracing pairs of from the fourth to sixth character p[4, 6] = 'bca' of the pattern p, we get (13, 20, 01) = (13, 20)(20, 01) and (14, 24, 07) = (14, 24)(24, 07). Thus, finally, we can see the searching pattern p = 'cabbca' in two places having the current sorting position numbers 21 and 22. Therefore, in cases that match any pattern of p's, we only have to search from the first position p[1] in order and for the length of the pat tern. In the case that matches the pattern with the original text parts by the usual simple method, without matching starting from the first position and up to the last position in order, we cannot make sure whether the pattern matches the text parts or not.

If we find the searching pattern in several places with the current sorting position numbers from the text, we often want to know the text parts before and behind the pattern characters. The characters continued behind of the pattern p can be obtained as well by using the decoding procedure. I explain about the case that finds before the characters of the pattern p. To the current sorting position number 21, seeking the current sorting position number of the previous sorting position number 21, we know the number 28. Namely, we get the current sorting position number 28 seeking the pair of (current sorting position number, previous sorting position number) = (28, 21). Therefore, the first character c of the cyclic shifts having the current sorting position number 28 is just one before the character of the pattern p. Thus, for the current sorting position number i, seeking for the current sorting position number x such that a pair of (current sorting position number, previous sorting position number) = (x, i), we can obtain the characters of just in front of the pattern p as necessary. In the computer screen, we can display the text part before and behind the characters of the pattern p.

position number	a 10	b 9	c 13
01	**02**	**13**	**01**
02	**05**	**14**	**03**
03	**11**	**20**	**04**
04	**12**	**24**	**06**
05	**15**	25	**07**
06	**16**	27	**08**
07	**17**	30	**09**
08	**18**	31	**10**
09	**19**	32	**21**
10	28		**22**
11			**23**
12			26
13			29

Fig. 7 Additional coding of the frequency of characters.

Now, the drawback of the searching method of block sorted compression data is that it is necessary for the first stage decompressing and decoding which restores the last column of the array completely. The following describes a method where it is not necessary to restore the last column of the array completely. As shown in Fig. 7, we consider the case, which is encoded so that we can know beforehand that the unchangeable count of each character a, b, and c which appears in the characters of any columns of the array are 10, 9 and 13. If we know the frequency of each character beforehand, in the rearrangement for the current sorting, we can know the previous sorting position number even if we process from the head of the characters y[1, n] in order. First, the head y[1] = 'c' is the previous sorting position number 01 and the column of the character 'c' begins from the current sorting position number 20 (= 10 + 9 + 1). y[2] = 'a', which is the previous sorting position number 02 enters into the current sorting position number 01. y[3] = 'c', which is the previous sorting position number 03 enters into the current sorting position number 21 (= 10 + 9 + 2). We assume that the current sorting operations of the characters y[1, n] were done up to the character 'b' appears first since the searching pattern is p = 'cabbca. 'b' first appears in the thirteenth because y[13] = 'b' for the character 'b' and enters into the current sorting position number 11 (= 10 + 1). Tracing matching the pair of (current sorting position number, previous sorting position number) with the pattern p using the information obtained so far, we get (21, 03, 11, 13) = (21, 03)(03, 11)(11, 13) and (22, 04, 12) = (22, 04)(04, 12) and can match up to 'cabb' and 'cab', respectively. After that, doing the sorting operations in order up to y[24] = 'b', we can match (21, 03, 11, 13, 20, 01) and (22, 0, 12, 4, 2, 07) and see that the pattern p = 'cabbca' cannot appear except in these two places without the current sorting. Because we can see that '**' in (current sorting position number, previous sorting position number) = (16, **) is not the current sorting position numbers from 11 to 18 corresponding to the character 'b' since we seek up to y[24]. Thus, we can end the operations at the position of y[24]. However, we need to do the current sorting operations to the end of the characters y[1, n] in the worst case.

So far, I have explained the processes to match from the first position of the pattern p. However, if the frequency of the first character p[1] of the pattern p increases, the first matching processing becomes frequent and the operations to narrow down the candidates also become frequent. To avoid this, selecting the characters having fewer frequencies in the pattern p, starting the matching operation from the position, and after narrowing down the candidates, it becomes effective to match and trace back to the character of just in front of the starting position. We can narrow down quickly to seek for from the first place of the pattern p than to seek for the several places of the pattern p simultaneously.

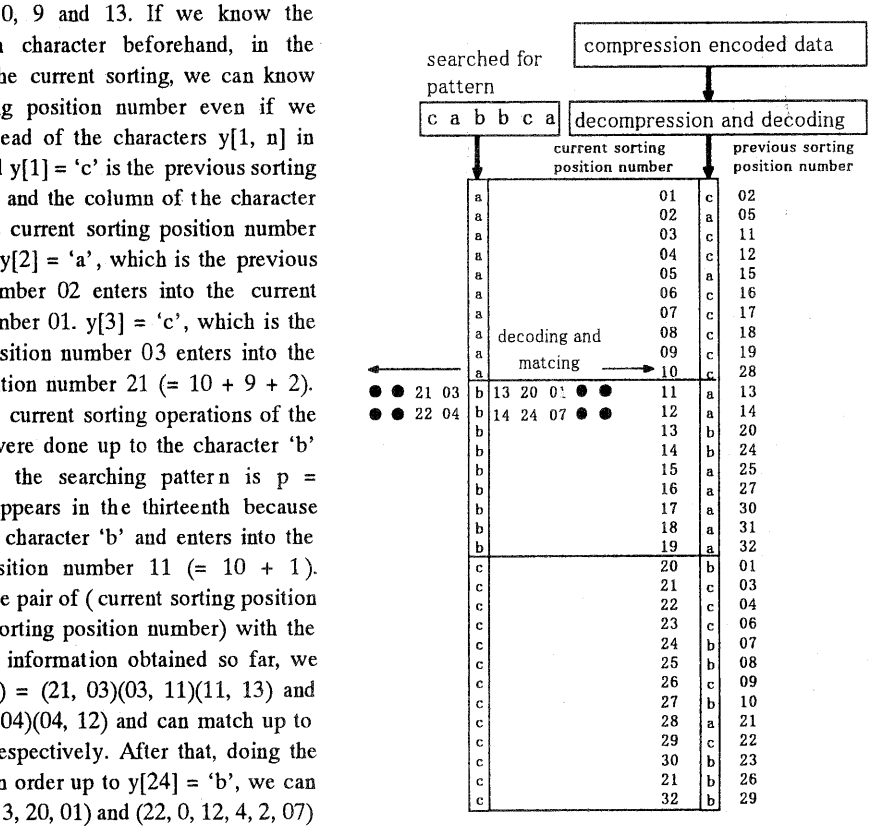

Fig. 8 Matching the pattern character having fewer frequencies in the text.

Although the current sorting operation of characters of the last column of the array is halfway, I showed the method can find the search pattern, but it is still necessary to decompress and decode in the first stage to some extent. Therefore, I consider a method where it is not necessary to decompress and decode in the first stage. I have already explained that it is important to find the one -to-one correspondence relation pair (current sorting position number, previous sorting position number) in the second stage

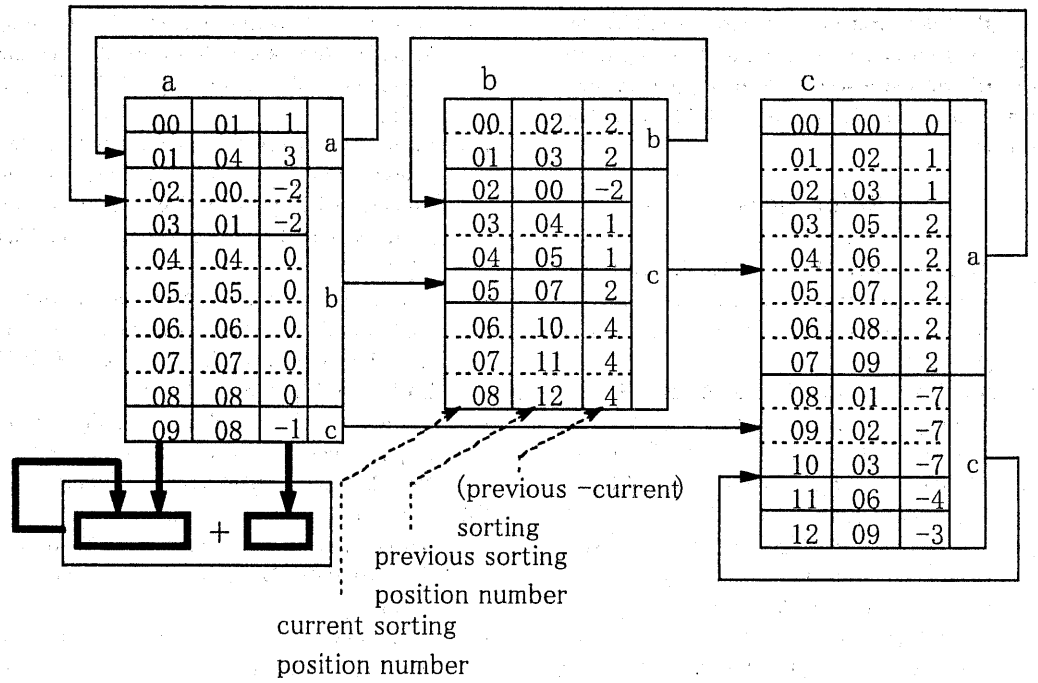

Fig. 9 Modified block sorting compression coding.

decoding of the block sorted compression data. If we can encode this corresponding relation in the preserving form, we can omit the corresponding operations when decoding. Now, as shown in Fig. 9, we notice that the previous sorting position number is also the consecutive number in the same way corresponding to the running appearance of the same character of y[1, n]. We also know that characters are occurred in the sorted order if we separate characters of the previous sorting position among running characters because they are sorted by nature. Moreover, finding the number that subtracts the current sorting position number from the previous sorting position number, it might be better for us to encode relatively. We can expect shorter length code if we encode relatively. That is, as shown in Fig. 10, instead of compressing and encoding the column of characters of y[1, 32], pointing to the beginning position in which the same sorted character appears, if we compress and encode so well that we can compute the previous sorting position number corresponding to it, we can directly execute the search (to some extent) as shown in Fig. 6. The symbol code 'i+j' shows i appears j+2 times. The symbols ( ) show simply pauses they do not encode.

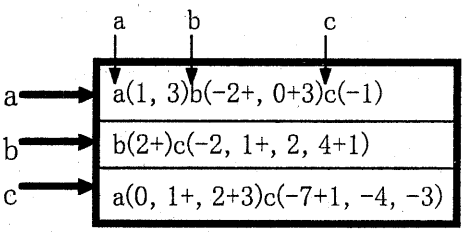

Fig. 10 Coding that can calculate the sorting position number.

4. Conclusion

An alternative to high-speed searches on a large quantity of compressed data is the block sorting compression algorithm for both compression and sorting tasks. The block sorting compression algorithm, a searching pattern locates in several places in the text appears in contiguous rows of the array since the whole text is sorted in lexicographic order with the array of the cyclic shifts, and it can decompress and decode from anywhere in the text

and with batch matching. I also modified the compression method so that it can decode directly and search without a second decoding process step, which is necessary in the original block sorting algorithm.

References

1) Amir, A., and Benson, G.: Efficient two dimensional compressed matching, Proc. of Data Compression Conferrence, pp. 279-284, (Mar. 1992).
2) Amir, A., Benson, G. and Farach, M.: Let Sleeping Files Lie: Pattern Matching in Z-compressed Files, Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 705-714 (1994).
3) Arnavut, Z. and Magliveras, S. S.: Lexical Permutation Sorting Algorithm, The Computer Journal, Vol. 40, No. 5, pp. 292-295, (1997).
4) Burrows, M. and Wheeler, D.J.: A block-sorting lossless data compression algorithm, SRC Research Report, 124 (May 1994).
5) Effros, M.: Universal Lossless Source Coding with the Burrows Wheeler Transform, IEEE Proc. of DCC'99, pp. 178-187, (1999).
6) Farach, M. and Thorup, M.: String Matcing in Lempel-Ziv Compressed Strings, ACM STOC'95, pp. 703-712, (1995).
7) Fenwick, P.M.: 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 /Tech Rep130. ps.
8) Ferragina, P. and Manzini, G.: Opportunistic Data Structures with Applications, Technical Report TR00-03, Dipartmento di Informatica, Universita di Pisa, (March 2000).
9) Gasieniec, L. and Rytter, W.: Almost optimal fully LZW-compressed pattern, Proc. Data Compression Conference (DCC'99), pp. 316-325, (Mar. 1999).
10) Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., Arikawa, S.: Multiple Pattern Matching in LZW Compressed Text, Proc. Data Compression Conf. DCC'98, pp. 103-112, (1998).
11) Lecroq, T.: Experimental Results on String Matching Algorithms, Software-Practice and Experience, Vol. 25(7), pp. 727-765, (Jul. 1995).
12) Manber, U.: A Text Compression Scheme that Allows Fast Searching Directly in the Compressed File, ACM Trans. on Information Systems, Vol. 15, No. 2, pp. 124-136, (Apr. 1997).
13) Michimoto, K. and Mashima, K.: Power of High-Speed Full Text Searching ('Zenbun Kensaku no Iryoku' in Japanese), NIKKEI BYTE, pp. 142-168, (1996.10).
14) Nakatsu, N.: An Alphabetic Code and Its Application to Information Retrieval , Vol. 34, No. 2, Trans. of IPSJ, pp. 312-319 (1993).
15) Sadakane, K.: A Note on the Compressed Suffix Arrays, In Summer DB Workshop, IPSJ, SIGDBS-122-7, pp. 49-56, (2000) (in Japanese).
16) Sedgewick, R.: Algorithms, 2nd ed., Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, U.S.A., (1988).
17) Stephen, G.A.: String Searching Algorithms, World Scientific, Singapore (1994).
18) Tokunaga, T.: Computer and Language Vol. 5, Information Retrieval and Natural Language Processing, University of Tokyo Press (1999) (in Japanese).
19) Uematsu, T.(植松友彦)：文書データ圧縮アルゴリズム入門, CQ Shyuppan, Tokyo (1994).
20) Welch, T.A.: A Technique for High Performance Data Compression, IEEE Comput., Vol. 17, pp. 8-19, (Jun. 1984).
21) Yokoo, H.: Data compression using a sort-based context similarity measure, The Computer Journal, Vol.40, No. 2/3, pp.94-102, (1997).
22) Yokoo, H.: Note on Block Sorting Data Compression, IEICE Trans., Vol. J81-A, No. 3, pp. 437-444, (1998).
23) Ziv, J. and Lempel, A.: A Universal Algorithm for Sequential Data Compression, IEEE Trans. on Inform. Theory, Vol. IT-23, No. 3, pp. 337-349, (May 1977).
24) Ziv, J. and Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding, IEEE Trans. on Inform. Theory, Vol. IT-24, No. 5, pp. 530-536, (Sep. 1978).