A New Chinese Text Compression Scheme Combining Dictionary Coding and Adaptive Alphabet-Character Grouping

Hung-Yan Gu

Department of Electrical Engineering, National Taiwan Institute of Technology
No. 43, Keelung Rd., Sec. 4, Taipei, Taiwan, ROC
Fax: 886-2-2730-1081, E-mail: guhy@mail.ntust.edu.tw

ABSTRACT

In this paper, a new scheme is proposed for Chinese text compression. The factors, compression rate and decompression speed, are specially considered in order to help such applications as full-text searching. Actually, our scheme is based on the LZ77 scheme. The modifications made include alphabet-augmenting to obtain better compression rate, and adaptive-grouping to have faster processing speed when facing large alphabet. Alphabet-augmenting is meant to place the 32 control characters defined here and about 6,000 frequently used Chinese characters into the alphabet while adaptive-grouping is meant to move a referenced alphabet-character dynamically to another character group that use less bits to encode it. The alphabet characters are divided into 8 groups initially. To implement adaptive-grouping, a new strategy is proposed and compared with the convention strategy of move-to-front. In this paper, different schemes compared are all programmed as ready-to-use software package, i.e. not in simulation. The experiments show that the proposed strategy for implementing adaptive-grouping can not only obtain significant compression rate improvements but also have much faster processing speed than the strategy, move-to-front. In addition, the compression rates obtained by our scheme are better, in 5.4% and 7.5%, than those obtained from the popular software package, ARJ, when two example files with text data from Chinese textbooks of primary school and editorials of newspaper are processed respectively.

Keywords: text compression, dictionary coding, large alphabet, adaptive character-grouping

1. Introduction

Several works on Chinese text compression have been published recently [1,2,3,4,5]. However, the aims of these works are not similar. One extreme case is that only the factor of compression rate (defined here as the output file length divided by the input file length) is concerned and computational limitations (needed memory space and computing time) are disregarded. Conversely, on the other cases, computational limitations are considered (i.e. practically usable compressor and decompressor software are implemented) and the three factors, compression rate, compression speed, and decompression speed, are treated equally or unequally. For example, we have proposed and implemented a Chinese text compression scheme before [5], and the factor of compression rate is given more weight.

In this paper, we propose a new scheme for Chinese text compression, which gives more weight to compression rate and decompression speed while slower compression speed is tolerated. The motivation for such kind of treatment to these factors is as the following. Consider the applications such as full text searching of large amount (e.g. more than 10 Mbytes) of Chinese text. The text data inevitably need to be stored in a secondary storage device (e.g. hard disk or compact disk). To increase the transmission speed from disk to main memory and to reduce the disk space required, the original text must be compressed before being stored into disks. However, this doesn't mean that only the compression rate is the important factor. Note that the compressed text must be restored to its original form before it can be searched. If the decompression speed is slow, the users would not accept such a full-text searching system. Therefore, the factor of decompression speed is also very important.

There are many compression schemes proposed in the literature, which can probably be used to compress Chinese text. Here, we roughly classify these schemes and call them character-predicting compression schemes and dictionary-searching compression schemes. In character-predicting schemes [1,3,5,6], a probability model is used to estimate the occurrence probabilities of the alphabet characters in order to predict what character would come next. On the other hand, in dictionary-searching schemes [7,8,9,10,11,12], a dictionary is maintained to support the operation of finding the longest word, in the dictionary, that is also a prefix of the input character string to be compressed. In general, dictionary-searching schemes are faster than character-predicting schemes. Therefore, in this paper, we decide to base the new scheme on the framework of dictionary-searching and then study how to improve compression rate. In fact, the compression scheme proposed is based on the LZ77 scheme [7,13] and has several notable modifications made. The two main modifications are:

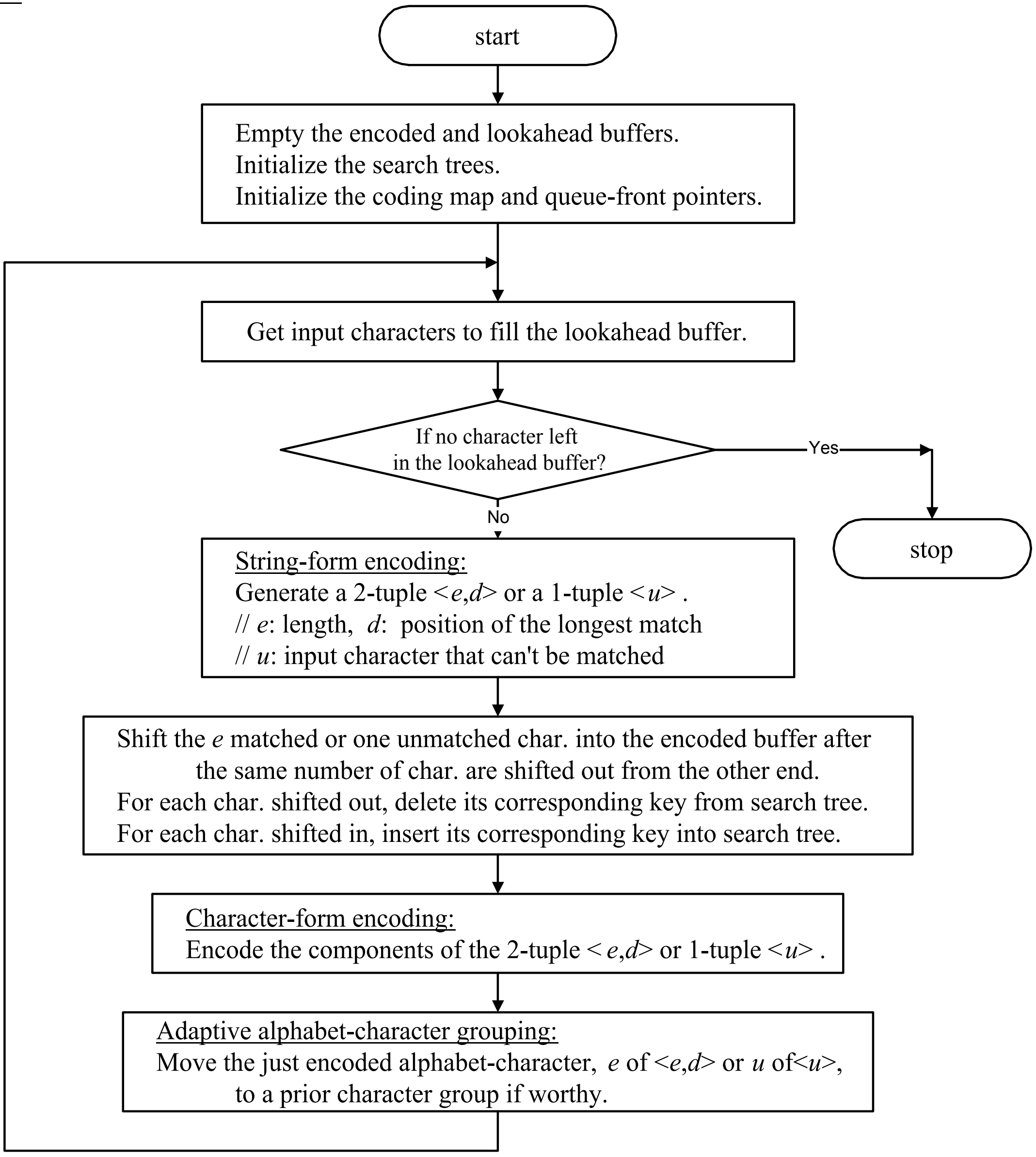
Alphabet-augmenting:

In addition to the 256 characters of 8 bits ASCII code, the most-frequently used 5,401 characters defined in Big-5 code and 32 control-characters defined here are also treated as alphabet characters in order that better compression rates can be obtained [5]. However, large alphabet size would induce implementation problems and result in slower processing speed. So, efficient implementation methods may be as valuable as the idea (using large alphabet) itself. Here, adaptive-grouping and its implementation strategy are proposed to overcome large-alphabet's problems.

Adaptive-grouping:

Initially, the alphabet characters are arbitrarily divided into 8 character groups. Then, accompanying the processing of compression or decompression, the frequently observed characters are moved to the groups whose elements are encoded with fewer bits while the seldom observed characters are moved in the reverse direction. Adaptive grouping of alphabet characters and its implementation strategy are proposed here to solve the problems incurred when using large alphabet, and to extend the scope of static-grouping studied by others [2, 12].

As an overview, the main processing flow of our compression scheme is drawn in Fig. 1. Since some of the operations (e.g. initialization and getting input characters) are trivial, they would not be explained in details. In the second section of this paper, the relevant



start
Empty the encoded and lookahead buffers.
Initialize the search trees.
Initialize the coding map and queue-front pointers.
Get input characters to fill the lookahead buffer.
If no character left in the lookahead buffer?
Yes
No
stop
String-form encoding:
Generate a 2-tuple <e,d> or a 1-tuple <u> .
// e: length, d: position of the longest match
// u: input character that can't be matched
Shift the e matched or one unmatched char. into the encoded buffer after the same number of char. are shifted out from the other end.
For each char. shifted out, delete its corresponding key from search tree.
For each char. shifted in, insert its corresponding key into search tree.
Character-form encoding:
Encode the components of the 2-tuple < e,d> or 1-tuple <u> .
Adaptive alphabet-character grouping:
Move the just encoded alphabet-character, e of <e,d> or u of<u>, to a prior character group if worthy.

Fig. 1 Main processing flow of our scheme.

matters about string-form encoding are described. LZ77 scheme is briefly reviewed, our modifications are explained, and the structure and maintenance of search trees are described. In the third section, the relevant matters about character-form encoding are described. The reason for dividing the alphabet characters into several groups is explained. Then, an effective strategy to implement adaptive-grouping is described. To test and compare our compression scheme with other schemes, ready-to-use compressor and decompressor programs are written. With these programs, compression experiments are conducted and their results are described in section four. Finally, a concluding remark is given in section five.

2. String-form encoding: character string matching and encoding

In dictionary-searching paradigm, the compressing process can be divided into two processing stages, i.e. string-form encoding (would be discussed in this section) and character-form encoding (would be discussed in the next section). About string-form encoding, the original LZ77 scheme and one of its variant, the LZSS scheme, would be briefly reviewed because our compression scheme is based on them. Then, the modifications made in our scheme are described.

2.1 Brief review of two relevant schemes

In the LZ77 scheme, a buffer consisted of two adjacent parts is used. Such a buffer is also called a sliding window. As illustrated in Fig. 2, the right part called the lookahead buffer is used to store the input character string that is to be compressed, and the left part

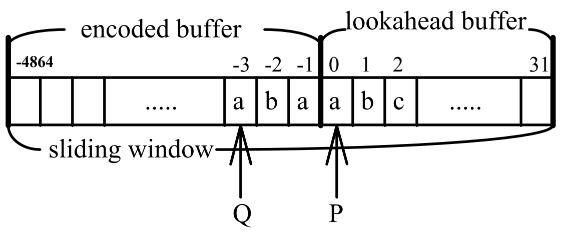


Fig. 2 An example of sliding window used in the LZ77 scheme

called the encoded-buffer is used to store the most-recently processed character string. The compression procedure of the LZ77 scheme can be explained with the three steps:

(1) Search the encoded buffer to find a longest match between a substring in the encoded buffer and a prefix substring in the lookahead buffer. That is, the encoded buffer is treated as the dictionary.

(2) Generate a 3-tuple <d, e, u> to encode the longest match. In the 3-tuple, the first component d record the longest match's position. In practice, d is the difference of the pointer P (pointing to the first character of the lookahead buffer) and the pointer Q (pointing to the first character of the longest match in the encoded buffer). Besides, the component e is the length of the longest match, and the component u is the first unmatched character in the lookahead buffer.

(3) Shift the entire sliding window e+1 characters to the right. That is, e+1 characters in the leftmost are shifted out and e+1 more characters are inputted and shifted into the rightmost. Such operations explain why the buffer is called a sliding window.

When the procedure is executed iteratively, a sequence of 3-tuples, $<d_1, e_1, u_1>$, $<d_2, e_2, u_2>$, ..., would be obtained. According to this procedure, it is apparent that an LZ77 compressor would spend much more time than its corresponding decompressor because the compressor must perform the time-consuming operations of substring matching while the decompressor need not. Therefore, decompressing speed is usually much faster than compressing speed.

Latter in the 1980s, Bell, et al., noted that the sequence of 3-tuples generated by the LZ77 scheme can be viewed as an alternative sequence of 2-tuples and 1-tuples, i.e. <d_1, e_1>, <u_1>, <d_2, e_2>, <u_2>, ... Accordingly, they argued whether 2-tuples and 1-tuples must appear alternatively. In fact, they had proposed a variant scheme called LZSS [9,13] that doesn't have such restriction. In the LZSS scheme, a cost criterion is used to determine whether a 2-tuple or a 1-tuple should be generated, and one more bit is placed at the front of the generated code to indicate whether it is a 2-tuple or 1-tuple. According to their experiments, the LZSS scheme can indeed obtain better compression rates than LZ77.

2.2 Modifications made in our scheme

After studying the LZSS scheme, we think that the use of an extra bit to indicate the following tuple's type can be replace by a more efficient way. In detail, we first change the order of a 2-tuple's two components, i.e. form a 2-tuple as <e, d> instead of <d, e>. Then, treat the e component as control characters (totally 32) and place them into the alphabet that a 1-tuple's component character comes from. So, this alphabet is similar to the alphabet of ASCII characters, which has also control characters. In this way, the extra bit is not needed because the decompressor can determine whether the currently decoded code unit is a 2-tuple or 1-tuple by checking the first component decoded. If a control character is decoded, it apparently comes from a 2-tuple and the second component should be decoded immediately. Otherwise, it is a 1-tuple and has only one component. This modification would result in better compression rate than the LZSS scheme when it is cooperated with the adaptive-grouping mechanism discussed in next section.

In addition to place the control characters defined here into the alphabet, we have also place the 5,401 most-frequently used Chinese characters defined in Big-5 code into the alphabet. However, the other characters in Big-5 code are still treat as two successive 8-bits ASCII characters. Why don't we put the entire Chinese characters defined in Big-5 code into the alphabet or let its size be smaller than 5,401? It is because in the preliminary experiments we found the compression rate would be slightly degraded about 0.5% when adding 4,096 more characters into the alphabet, and would be degraded about 2.4% when the last 2,048 characters of the adopted alphabet are removed. Another question may be asked is why large alphabet (e.g. 6,000 characters) is better in compression rate than small alphabet (e.g. 256 characters). We think it is due to the very uneven use of Chinese characters. For an example, consider the two probability distributions, {1/2, 1/2} and {1/32, 1/32, 3/32, 27/32}, representing characters' occurrence probabilities of a small alphabet and a large alphabet respectively. It can be seen that the latter distribution's entropy, $-\log_2(1/32) - \log_2(1/32) - \log_2(3/32) - \log_2(27/32) = 0.8395$ bit, is smaller than the former distribution's entropy, 1 bit. So, a large alphabet instead of a small one is adopted here. When using a large alphabet to compress Chinese text, it is apparent that the inputted data bytes can't be directly put into the lookahead buffer of Fig 2. Instead, the data bytes have to be lexically scanned beforehand to convert each observed character (Big-5 characters is checked prior to ASCII characters) into a corresponding integer number representing it. Then, these numbers are put into the buffer.

About the problem of string match, the data structures of hashing and ordinary binary search tree [14,15] are combined in order to obtain fast searching speed. In detail, when a character x is newly shifted into the encoded buffer from the lookahead buffer, x and its 31 successor characters together would be considered as a key and inserted into one of the 512 search trees maintained. The determination of which search tree to insert is by means of a hashing function that map a key's first two characters to a value between 0 and 511. Here, the hashing function is, $h(x_0..x_{31}) = (127*x_0 + x_1) \% 512$, where '%' is the modulus operator. On the contrary, when a character x is to be shifted out of the encoded buffer, the key comprised of x and its 31 successor characters is deleted from the search tree determined by the same hashing function. In terms of hashing and the 512 search tree maintained, the longest match between a prefix substring in the lookahead-buffer and a substring anywhere in the encoded buffer can be found with very fast speed. The matching procedure is simple. First, the key comprised of the 32 characters in the lookahead buffer is hashed by the same function to select a search tree. Then, the length of the longest match is recorded when walking down the search tree. Because the first two characters of a key is used by the hashing function to determine a search tree, the length, 0, of the longest match doesn't mean that the first character of the lookahead buffer cannot be matched, i.e. the value 0 may mean no match or matched a character. So, we treat both the length values, 0 and 1, as unsuccessful match, and a 1-tuple will be sent out by the string-form encoding stage.

3. Character-form encoding : tuple component encoding and adaptive character grouping

Each time when the string-form encoding stage outputs a 2-tuple, <e, d>, or a 1-tuple, <u>, the character-form encoding stage will follow to encode the components of this tuple. Then, it will decide whether the alphabet character corresponding to e or u of the just encoded tuple should be move to another character group to perform adaptive grouping of alphabet characters.

3.1 Tuple component encoding

When encoding the first component of a tuple (1-tuple or 2-tuple), a code consisted of two parts, i.e. the group code and element code, will be generated. This is because the alphabet characters have been divided into 8 character groups and each character may be dynamically moved to another group, i.e. a character is not always an element of a specific group. The reason to divide the alphabet's characters into several character groups is as the following explanation. In some conventional data compression schemes, the alphabet characters are imagined as being placed in a stack and the MTF(move-to-front) strategy [11, 12, 13] is used each time after encoding an input character to move the referenced alphabet character to the top of the stack. Since the alphabet adopted here has so many characters, the processing speed would become very slow if the MTF strategy is directly programmed, i.e. the goal of fast decompression speed cannot be achieved. Therefore, we study to divide the alphabet characters into several groups and use a new adaptation strategy to move the alphabet characters dynamically among these groups, i.e. adaptive-grouping, in order that both the faster decompression speed and better compression rate can be obtained. The details of the new adaptation strategy is described in section 3.2.

In this paper, we use variable-length code to represent and distinguish different character groups but the element characters of a group are represented by fixed-length code. In detail, the 6,827 characters of the alphabet adopted are divided into the 8 fixed-size groups:

Group 0: of 1 element, group code: $(010)_2$, need no element code.
Group 1: of 2 elements, group code: $(100)_2$, element code length: 1 bit.
Group 2: of 8 elements, group code: $(1110)_2$, element code length: 3 bits.
Group 3: of 32 elements, group code: $(101)_2$, element code length: 5 bits.
Group 4: of 128 elements, group code: $(011)_2$, element code length: 7 bits.

Group 5: of 512 elements, group code: $(00)_2$, element code length: 9 bits.
Group 6: of 2048 elements, group code: $(110)_2$, element code length: 11 bits.
Group 7: of 4096 elements, group code: $(1111)_2$, element code length: 12 bits.

The determination of the 8 groups' size is according to the compression rates obtained in the preliminary experiments for a few combinations of the groups' sizes under the constraints, the number of character groups is fixed at 8 and the summation of the 8 groups' sizes must be greater than 5689 (32 match-length control characters, 256 ASCII characters, and 5401 most-frequently used Big-5 Chinese characters).

For the encoding of the second component of a 2-tuple, <e, d>, a code consisted of three parts, i.e. the group code, element code, and remainder code, will be generated by the character-form encoding stage. The encoding procedure is :

(a) Divide d by 64 to obtain the quotient dq and the remainder dr. **(** The value range defined for dq is 0 ~ 75 and is obviously 0 ~ 63 for dr. So the value range for d is 0 ~ 4863.**)**

(b) Determine the group of which the value of dq is an element. Then, generate a variable-length group code to represent this group, and generate a fixed-length element code to indicate which element of this group dq is.

(c) Represent dr by a fixed-length code of code length 6 bits.

Here, we divide the possible values of dq into 5 groups according to the results of the preliminary compression experiments. In details, the values 0~3 are put into the first group and are assigned the group code $(00)_2$. The values 4~11 are put into the second group and are assigned the group code $(110)_2$. The values 12~27 are put into the third group and are assigned the group code $(01)_2$. The values 28~43 are put into the forth group and are assigned the group code $(111)_2$. The values 44~75 are put into the fifth group and are assigned the group code $(10)_2$. We had once tried to dynamically move the value of dq among the 5 groups similar to adaptive grouping of the alphabet characters, but the compression rate difference is within 0.05%. So, we choose to group the possible values of dq statically only.

3.2 Adaptive alphabet-character grouping

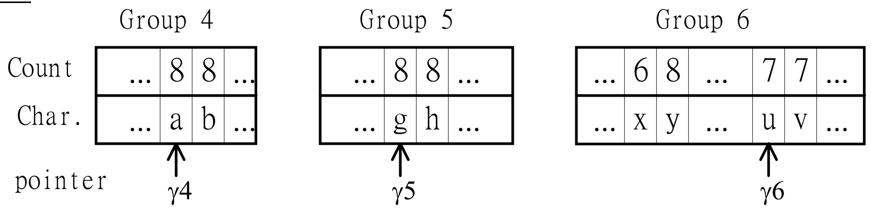
When starting to run, the compressor (or decompressor) has no knowledge about each alphabet character's occurrence probability. Besides, it is found that initial grouping has nearly no influence on compression rate according to the preliminary experiments conducted. So, the initial grouping here is just to put lower numbered character into lower numbered group. Then, accompanying the processing of compression or decompression, the more frequently observed alphabet characters are moved, according to an adaptation strategy's directing, to the group that use less bits to encode its elements. This idea of adaptive grouping of alphabet characters can result in compression rate being improved a lot. To obtain the best compression rates, we had tried several adaptation strategies to adjust a just encoded (or decoded) alphabet character's staying group. Finally, a strategy is found to be more effective and has smaller time overhead. In this strategy, a counter variable N(s) is associated with a alphabet character s to count the times it is observed, and each of the 8 character groups is treated as a queue such that its elements' entering and leaving is controlled in a first-in-first-out manner. In details, suppose the character s is just encoded and is currently an element of the group X. Then, the proposed strategy can be illustrated by the following procedure:

```
(1)   Z <= X;                   /* Record the original group number into variable Z */
(2)   N(s) <= N(s) + 1;         /* Update the occurrence frequency of the character s */
(3)   Y <= X - 1;               /* Set variable Y to denote the group immediately prior to X */
(4)   while( X≠0  and  N(s)>N(front(Y) )  {
```

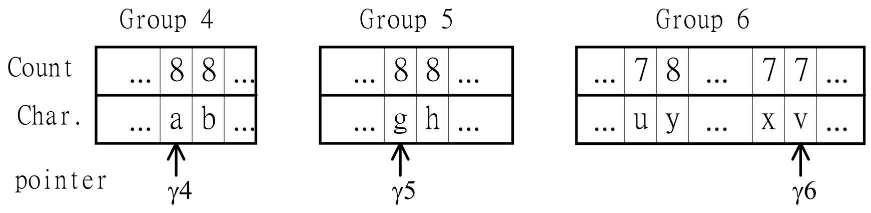
```
(5)        w <= dequeue(Y);          /*  Remove a character from the front end of queue Y  */
(6)        if(X=Z) replace(s, w);    /*  Replace X's element character s by the character w  */
(7)        else  enqueue(X, w);      /*  Insert character w to the rear end of queue X  */
(8)        X <= Y;                   /* Set variable X to denote the group previously denoted by Y  */
(9)        Y <= X - 1;               /*  Set variable Y to denote the group immediately prior to X  */
(10)  }
(11)  if(X=Z)  {                     /*  if character s cannot be moved to another group  */
(12)       w <= dequeue(X);
(13)       replace(s, w);
(14)  }
(15)  enqueue(X, s);
```

In this procedure, the function, front(Y), is used to lookup the character at the front end of the queue Y, the functions, enqueue(X, w) and dequeue(X), are as explained in the comment part, and the function, replace(s, w), is used to place the character w into the queue element position currently occupied by the character s. From the forth to the tenth lines of this procedure, it can be seen that our strategy will move a just encoded character from a latter group to a former group repeatedly if its occurrence frequency is found to be larger than the frequency of the prior group's queue-front character. Furthermore, from the eleventh to the fourteenth lines, our strategy will move the just encoded character to the rear end of its current staying queue if it cannot be moved to a prior queue. The purpose is to reduce the probability of being immediately kicked out to a latter queue when the following characters are adapted in the same way.

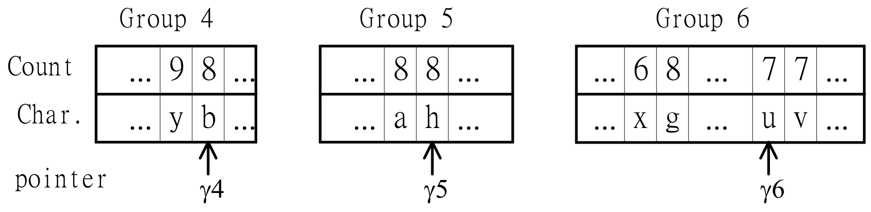
To explained our strategy more solidly, example queue situations before and after adapting a alphabet-character are drawn in Fig. 3. Fig. 3(a) illustrates the situation before performing adaptation of a just encoded character. First, suppose the just encoded character is the character 'x' in group 6. Because its occurrence count 7 (after incremented) is smaller that 8 of the queue-front character 'g' of group 5, it would not be move to the prior group. Nevertheless, it is interchanged with the queue-front character 'u' of its current group, and the queue-front pointer is advanced to make it as the queue-rear character. This results in the situation shown in Fig. 3(b). Secondly, suppose the just encoded character is the character 'y' in group 6. Since the occurrence count 9 (after incremented) of 'y' is now greater than 8 of the queue-front character 'g' of group 5, 'y' is therefore interchanged with 'g' and the queue-front pointer of group 5 is advanced to make 'y' as a queue-rear character. Then, it is tried repetitively to move 'y' to a prior group. _____



(a) Queue situation before adapting an encoded character.



(b) Queue situation after adapting the character x.



(c) Queue situation after adapting the character y.

Fig. 3 Example queue situations to explain our adaptation strategy.

Suppose the occurrence count of 'y' is greater than the count of the queue-front character 'a' of group 4 but is smaller than the count of the queue-front character of group 3. So, finally, 'y' would be placed at the position previously occupied by 'a' and 'a' is placed at position previously occupied by 'g', and the queue-front pointers γ4 and γ5 are both advanced. This results in the situation shown in Fig. 3(c).

The strategy proposed not only can obtain lower compression rate but also is efficient and easy to implement. The data structure used is as shown in Fig. 4, which is called coding map in Fig. 1. In this figure, when a character is to be encoded, the corresponding α pointer can be followed to find the character group of which it is an element and the order number within the group. On the contrary, when a group number and an element

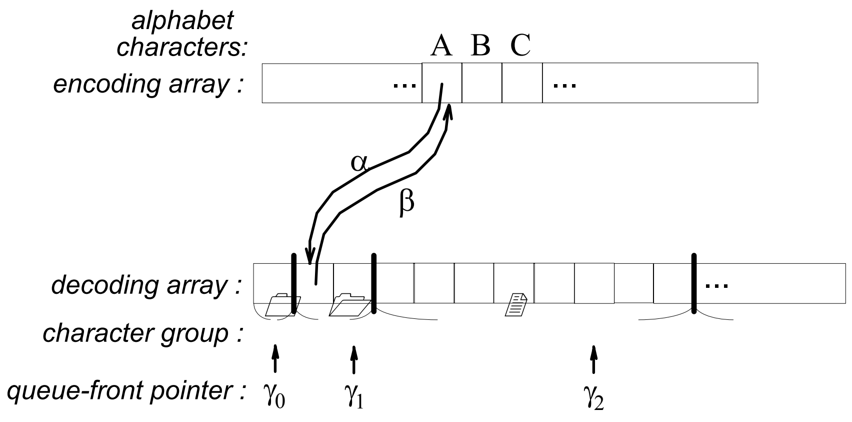


Fig. 4 Data structure for the proposed strategy

number are obtained after decoding, the corresponding β pointer can be followed to find the original character. To support the proposed strategy, each logically formed character group, represented on the decoding array, can be treated as a circular queue and therefore only a queue-front pointer is needed to implement the functions, enqueue(X,w), dequeue(X), and front(X).

4. Test experiments and results

In this section, the newly proposed strategy (for adaptive grouping of alphabet characters) is compared with the conventional strategy of move-to-front (for adjusting alphabet character's position) within the framework of the proposed compression scheme. Then, the compression scheme using the new strategy is compared with a popular compression package, ARJ. Note that all comparisons are made according to experiment results. So, the compression schemes, using the new strategy or the conventional MTF strategy, are both programmed, with C language, as ready-to-use software packages. Then, the compression rate can be directly computed as the output file length divided by the input file length.

In the experiments, four test text file, named CX1, CX2, CX3, and CX4, are used. The text in CX1 are collected from the twelve Chinese textbooks of primary school and have totally 231,737 bytes. The CX2 file contains 115 articles of newspaper editorials (for adults) and has totally 170,127 bytes. Each article's length is between 1k and 2.3 bytes. Besides, the text in CX3 are collect from the first ten chapters of the classical novel, The Dream of the Red Chamber, and have totally 140,628 bytes, while the CX4 file contains the text of a modern novel, An Air Hostess written by Shuo Wang, and has totally 62,584 bytes. As the hardware platform, a 486DX4-100 personal computer operated by MS-DOS operating system is used.

4.1 Comparison with MTF strategy

Here, the compression scheme using the newly proposed strategy is called LZG while the one using the MTF strategy is called LZM. That is, we intend to compare the two strategies while the other conditions are kept the same. For LZM, we choose to use the variable-length code derived from the group-element code proposed in section 3.1 to encode a character's current position within the alphabet characters' stack. In detail, the position numbers, 0, 1, 2, 3, 4, 5, 6, ..., are encoded as $(010)_2$, $(100,0)_2$, $(100,1)_2$, $(1110,000)_2$, $(1110,001)_2$, $(1110,010)_2$, $(1110,011)_2$, ..., respectively. There are two considerations to use this code. First, just the two adaptation strategies themselves are intended to be compared so it's better to let them use the same code. The second, in preliminary experiments for LZM scheme using either large or small alphabet, the group-element code proposed here is found to obtain very significant improvement (compression rate decreased more than 8%) over the commonly used variable-length code. In such variable-length code, the position numbers, 0, 1, 2, 3, 4, 5, 6, ..., are encoded as $(1)_2$, $(001)_2$, $(011)_2$, $(00001)_2$, $(00011)_2$, $(01001)_2$, $(01011)_2$, $(0000001)_2$, ..., respectively [13].

In addition, the factor of using large or small alphabet is also considered to test the two strategies. For small alphabet, only 256 ASCII characters plus 32 control character representing string match-length are included, i.e. Big-5 Chinese characters are not checked and all input data are treated as ASCII characters. This condition is indicated by '-S' in LZG-S and LZM-S while the condition of using large alphabet is indicated by '-L' in LZG-L and LZM-L. Note that the two strategies would both obtain worse compression rates (especially the MTF strategy) when the character groups' sizes proposed in section 3.1 are directly used under the small alphabet condition. Therefore, in preliminary experiments, a few combinations of character groups' sizes are tried and it is found that the groups' sizes had better be set to 2, 8, 8, 16, 32, 32, 64, and 128, with the group codes defined as, $(00)_2$, $(01)_2$, $(1100)_2$, $(100)_2$, $(101)_2$, $(1101)_2$, $(1110)_2$, and $(1111)_2$, respectively. So, these group sizes and codes are used for the small alphabet condition.

After the compressor and decompressor programs being executed for different combinations of strategies, alphabet sizes, and test text files, the results as listed in Table I(a) and I(b), for compression rates and spent processing time respectively, are obtained.

From Table I(a), it can be seen that the scheme LZG-L (using the newly proposed adaptation strategy and large alphabet) has the best compression rates among the four schemes tested. In addition, the claim that large alphabet is much better (more than 10%) in compression rates than small alphabet for Chinese text compression is verified by the experiment results when comparing the first and third columns with the second and forth columns, respectively. Furthermore, it is also verified that the proposed adaptation strategy is significantly better (more than %4) in compression rate than the conventional strategy of MTF when comparing the third and forth columns with the first and second columns, respectively. We think this improvement is due to the incremental movement of a referenced alphabet character from a latter group to a former group. In contrast to our strategy, the MTF strategy directly moves a referenced character to the first position no matter its original position.

Table I Compression results for different combinations

Table I(a) Rates for different combinations of strategies, input data, and alphabet sizes

rates %	LZM-S	LZM-L	LZG-S	LZG-L
CX1	**61.5**	**51.2**	**57.3**	**47.3**
CX2	**72.1**	**57.8**	**66.5**	**53.0**
CX3	**71.2**	**58.4**	**66.0**	**54.1**
CX4	**70.7**	**57.6**	**65.7**	**53.5**

Table I(b) Time spent for different combinations of strategies, input , and alphabet sizes

time: sec.	LZM-S	LZM-L	LZG-S	LZG-L
CX1	**5.88, 1.54**	**12.79, 10.16**	**5.34, 0.88**	**3.35, 0.79**
CX2	**4.45, 1.35**	**10.82, 8.94**	**3.74, 0.73**	**2.27, 0.62**
CX3	**3.48, 1.04**	**10.07, 8.46**	**3.24, 0.60**	**1.81, 0.53**
CX4	**1.54, 0.49**	**5.02, 4.38**	**1.35, 0.27**	**0.82, 0.23**

About the time spent by the four compression schemes, the experiment results are listed in Table I(b). In each cell of this table, there are two numbers. The left is the time, in seconds, used by the compressor program while the right is the time used by the decompressor program. From this table, it is apparent that the scheme LZG-L not only has the best compression rates but also is the fastest among the four schemes no matter whether the concerned factor is compressing or decompressing speed. So, the compression scheme LZG-L is the representative of this paper and is recommended for practical use. Also, it can be found that the proposed adaptation strategy not only obtains better compression rates but also is faster than the conventional strategy of MTF when the third and forth columns are compared with the first and second columns, respectively, for small and large alphabet conditions. Besides, another fact can be seen when the time spent, in the third and forth columns, by the compressor and decompressor are compared, i.e. the time used by the decompressor is less than one third of the time used by the corresponding compressor. This is just what intended for. However, for LZM-L, the time used by the decompressor are compatible with the time by the corresponding compressor.

4.2 Comparison with popular compression package

Here, the scheme LZG-L explained in section 4.1 is compared with a popular software package, ARJ. After the four test text files are compressed, the compression rate results as listed in Table II are obtained. The second column is a copy of the forth column of Table __

Table II Compression rates for different combinations of schemes and input text

rates %	ARJ	LZG-L
CX1	**52.7**	**47.3**
CX2	**60.5**	**53.0**
CX3	**61.3**	**54.1**
CX4	**60.6**	**53.5**

I(a). From this table, it can be seen that the compression rates obtained from our scheme are at least 5.4% lower than the rates from ARJ. To be noted, such compression rate improvement is not for long text file only, i.e. it is independent of file length. This is illustrated in Fig. 5. In this figure, the compression rate is dynamically measured, at a few points within the test files, as the number of outputted data bytes divided by the number of inputted data bytes. According to the two pairs of nearly paralleled curves, in Fig. 5, for the test files CX1 and CX2, it is claimed that our scheme will obtain an almost constant improvement of compression rate even for Chinese text file of much shorter length.

About the processing speed, the compressor program for LZG-L can process, in average, 73.3 Kbytes per second according to the data in Table I(b). On the other hand, the decompressor program for LZG-L can process, in average, 278.8 Kbytes per second.

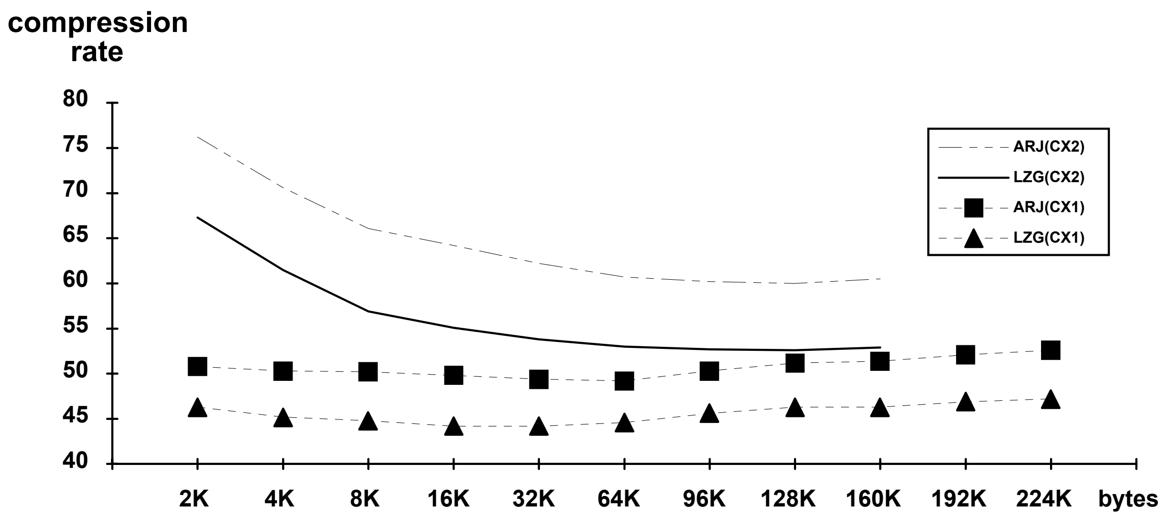


Fig. 5 Compression rates plotted against positions in the input text

Although the speed, 278.8 Kbytes/sec., is less than 540.2 Kbytes/sec. achieved by ARJ, it is still fast enough for practical use. Also, note that it is unknown whether ARJ is written in assembly language and/or highly optimized.

5. Concluding remark

In many applications such as full-text searching and on-line help facility, it is needed to compress Chinese text data to save storage space, and to decompress it fast to reduce user's waiting time. So, in this paper, a new Chinese text compression scheme is proposed with both compression rate and decompression speed being specially considered. This scheme is based on the LZ77 scheme. The main modifications made are alphabet augmentation and adaptive character grouping. We think large alphabet is the key factor that results in better compression rates of our scheme as compared to the popular package ARJ. On the other hand, under the condition of large alphabet size, the idea of adaptive-grouping and its implementation strategy let our scheme's processing speed be kept fast enough for practical use. The new strategy proposed for implementing adaptive-grouping is also shown to be significantly better, in compression rate, and faster than the conventional adaptation strategy, move-to-front, no matter whether the alphabet size is small or large. Besides Chinese text, we think the compression scheme proposed here is also applicable to the text data of other oriental languages such as Japanese, Korean, etc.

Reference

[1] L. Y. Tseng and T. H. Huang, "A Predictive Coding Method for Chinese Text File Compression", Journal of Computer, Vol. 2, No. 3, 1990, pp. 18-23.

[2] C. C. Chang and W. H. Tsai, "A Data Compression Scheme for Chinese-English Characters", Computer Processing of Chinese and Oriental Languages, Vol. 5, No. 2, 1991, pp. 154-182.

[3] M. K. Tsay, C. H. Kuo, R. H. Ju, and M. K. Chou, "Modeling for Chinese Text File Compression", Proceedings of the Second IASTED International Conference, Alexandria, Egypt, 1992, pp. 205-207.

[4] G. H. Ong and S. Y. Huang, "Compression of Chinese Textual Data Using a Restricted Variable Length Coding Scheme", Proceedings of International Computer Symposium, Hsin-chu, Taiwan, 1994, pp. 408-413.

[5] H. Y. Gu, "Adaptive Chinese Text Compression Using Large Alphabet, Markov Modeling, and Arithmetic Coding", Proceedings of National Computer Symposium, Chia-yi, Taiwan, 1993, pp. 568-575.

[6] I. H. Witten and T. C. Bell, "The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression", IEEE trans. Information Theory, Vol. 37, No. 4, 1991, pp. 1085-1094.

[7] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE trans. Information Theory, Vol. 23, No. 3, 1977, pp. 337-343.

[8] J. Ziv and A. Lempel, "Compression of individual sequence via variable-rate coding", IEEE trans. Information Theory, Vol. 24, No. 5, 1978, pp. 530-536.

[9] T. C. Bell, "Better OPM/L Text Compression", IEEE trans. Communications, Vol. 34, No. 12, 1986, pp. 1176-1182.

[10] T. A. Welch, "A Technique for High-performance Data Compression", IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.

[11] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme", Communication of the ACM, Vol. 29, No. 4, 1986, pp. 320-330.

[12] K. C. Chang and S. H. Chen, "A New Locally Adaptive Data Compression Scheme Using Multilist Structure", The Computer Journal, Vol. 36, No. 6, 1993, pp. 570-578.

[13] T. C. Bell, J. G. Cleary, and I. H. Witten, Text Compression, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.

[14] M. A. Weiss, Data Structure and Algorithm Analysis in C, Benjamin/Cummings Publishing Company Inc., Redwood, California, 1993.

[15] E. Horowitz, S. Sahni, and S. Anderson-Freed, Fundamentals of Data Structure in C, Computer Science Press, New York, 1993.