Large-alphabet Chinese Text Compression Using Adaptive Markov Model and Arithmetic Coder

Hung-yan Gu [1]

ABSTRACT

In this paper, an approach to compress Chinese text is proposed. This approach first extends the alphabet to include those Chinese characters in Big-5 code. Then, an adaptive Markov model is used to model the contextual dependency, and arithmetic coding is used to encode the data more compactly. In the case of large alphabet size, a practical implementation method for adaptive Markov model is studied, and a data structure for fast arithmetic coding is proposed. Furthermore, to demonstrate its credibility, this approach has been programmed into a ready-to-use software package.

A series of experiments have been conducted. The two example texts used are CX1 (from Chinese textbooks of primary school) and CX2 (from newspaper editorials). The experiments show that the compression ratios (output file length divided by input file length) obtained from our approach to be 42.3% and 48.4% for CX1 and CX2 respectively, which are considerably better than 52.8% and 60.6%, from the popular software package ARJ, and 53.0% and 60.8% from the package PKZIP. And for shorter files, our approach is still capable of obtaining a much improved compression ratios.

Keywords : data compression, arithmetic coding, Markov model, Chinese text

1. Introduction

The purposes of data compression are to save storage space, to reduce transmission time, and/or to protect secret information from non-authorized access. Since different types of data have very different characteristics, it cannot be expected that a certain data compression approach would always work to its highest efficiency. For example, text data compression requires that the restored data are the exact duplicate of the original, i.e. distortion free. However, some slight distortions are tolerable for the restored data of speech, image, or graphics as long as the information carried are not lost. In this paper, the pertinent data type is Chinese text, and compression efficiency is measured in terms of compression ratio, i.e. the size of the compressed data file divided by the size of the original data file.

---

[1] Manuscript received on June 29, 1994; revised April 13, 1995. The author is with the Department of Electrical Engineering, National Taiwan Institute of Technology, Taipei, Taiwan, Republic of China. E-mail address: root@guhy.ee.ntit.edu.tw .

For Chinese text compression, a few coding schemes have been proposed [1,2,3]. They have either been studied in simulation (have no practical application) or found to be lacking in compression efficiency. In addition, there are some popular and commercially available software packages (e.g. PKZIP and ARJ) that can be used to compress Chinese text to obtain certain space saving. However, the obtained compression ratios can be lowered further if the knowledge about Chinese text is applied. It is certain that these general-purpose compression packages have not take into account the arrangement of the code (i.e. Big-5 code commonly used in our nation) used in representing Chinese characters. In Big-5 code, each character is represented by two bytes. The first byte has a value from 161 to 254, while the second byte has a value from either 64 to 126 or 161 to 254. By taking such code arrangement and other factors into accounts, the approach proposed here can further lower the compression ratios by about 10%. Actually, this approach has been verified by writing both the compressor and expander (restore the compressed data file to the original data file) programs as ready-to-use software packages. Therefore, our approach is more believable than those approaches studied in simulation with impractical assumptions. Also, the compressor and expander programs developed can be used to process Chinese text file. The processing speed is more than 6K bytes per second on our machine (486-33 PC) while the basic requirement of main memory size is only 400K bytes.

A statistical data compressor (e.g. the one developed here) can be considered as comprised of two components, i.e. the model and the encoder [4]. The function of the model component is to provide a most likely probability distribution of the symbols in the alphabet. If the probability distribution is more accurately estimated, the encoder component would assign a more appropriate code to each symbol and thus obtain a lower compression ratio. According to the manner of estimating probability distribution, a model can be classified as adaptive, semi-adaptive, or static [4]. For an adaptive model, before encoding an input character, the probability distribution is dynamically adjusted to the characters seen so far. For a semi-adaptive model, the entire sequence of characters is inspected beforehand to estimate a single probability distribution, and then this distribution is provided to the encoder to encode all of the characters. For a static model, the probability distribution is always fixed regardless the characteristics of the text file being processed. In this paper, the adopted model, i.e. adaptive Markov model, is obviously an adaptive model.

The encoder component in a statistical compressor performs the main function of data compression. In each processing cycle, it gets a character from the input, encodes this character according to the probability distribution provided by the model component, and sends the compressed data bits to the output. There are a few techniques available for the implementation of the encoder. The earliest and best known one is Huffman coding [5,6] which can also be used with adaptive models [7]. However, Huffman coding only performs optimally if all the symbols' probabilities are integral power of 1/2, which is not normally the case in practice. In the late 1970s, the technique of arithmetic coding was discovered [4,6,8,9]. This is a breakthrough as arithmetic coding is free of the restriction of Huffman coding, i.e. each character must be encoded in integral number of bits. Therefore, this technique was adopted in this paper. In addition to arithmetic coding, another breakthrough is the discovery of Ziv-Lempel coding [4,6] that does not separate the model component from the coding component explicitly.

In the following section, the developed compressor and expander would be introduced in block diagrams. Then, in Section 3, the detail of adaptive Markov modeling is described. And the problems in practical implementation are discussed and resolved. In Section 4, the idea of arithmetic coding is reviewed briefly, and a data structure to accelerate the coding process is proposed. A series of experiments have been conducted, which would be described and discussed in Section 5. Finally, a concluding remark is given in Section 6.

2. Block Diagram of the Compressor and Expander

The block diagram of the developed compressor and expander programs is shown in Fig. 1. It can be seen from the diagram that the approach proposed is to use an adaptive Markov model

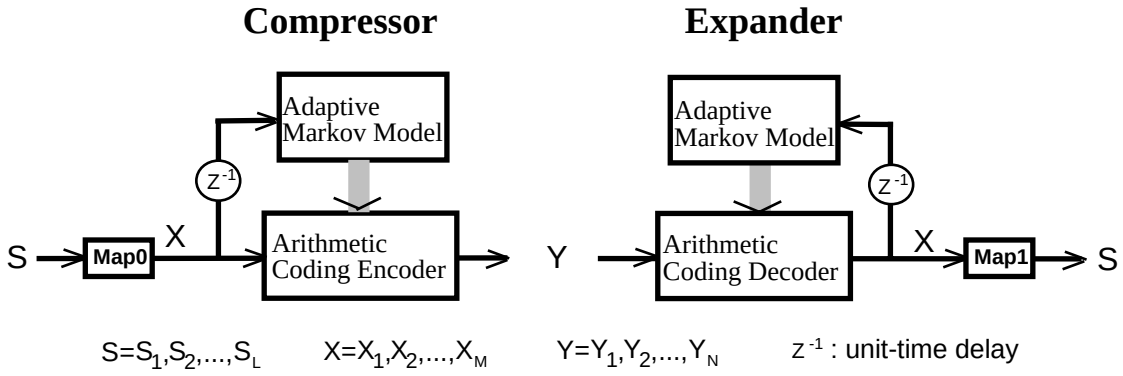

Fig. 1 The block diagram of the compressor and expander.

for the model component, apply the technique of arithmetic coding to the coder components (encoder and decoder), adopt an alphabet consisting of meaningful linguistic symbols (e.g. Chinese characters and English alphabet) instead of meaningless storage symbols (e.g. a byte's 256 patterns), and use the "Map#" blocks to convert between a symbol's different representations. At first glance, it may seem easy to put these ideas together. In reality, there are a few implementation problems which must be resolved for the approach to be practical. The implementation problems would be discussed in the following sections.

In Fig. 1, $S = S_1, S_2, ..., S_L$, represents the original text data (to be compressed or expanded to) as a sequence of bytes in memory. The block Map0 in the compressor functions as a lexical analyzer to scan and segment S into tokens according to the alphabet adopted. A token (like a random variable) can appear to be any single symbol of the alphabet as long as it is found in the prefix of S. For each token found, Map0 also translates it into an integer before outputting it. This integer is the order number of the token's instance symbol in the alphabet. Map0 is required because the alphabet adopted include both the ASCII and Chinese characters, and a Chinese character is represented in two bytes, which needs to be distinguished from the ASCII characters. To be precise, the alphabet is consisted of 256 ASCII characters, 471 graphical characters in Big-5 code, 5,401 commonly-used and 7,650 next-frequently-used Chinese characters in Big-5 code, and the pseudo control character called end_of_coding (automatically inserted as the last token by the compressor to signal the expander when to terminate processing).

In Fig. 1, $X=X_1,X_2, ..., X_M$ represents a sequence of integers as the output of Map0. For convenience of description, these integers would not be distinguished from the symbols they represent. That is, an integer u representing the u'th symbol of the alphabet will be referred to as the symbol u, and an integer token X_i will be referred to as the token X_i. In Fig. 1, the symbol Z^{-1} represents a unit delay buffer to prevent the current token to be encoded or just being decoded from being used by the model block. The sequence, $Y=Y_1,Y_2,...,Y_M$, represents the compressed data in a bit string format. At the rightmost position, Map1 performs the inverse function of Map0.

In Fig. 1, the two adaptive Markov models in the compressor and the expander provide the same probability distribution independently yet synchronously, i.e. no parameters about probability distribution needs to be transmitted from the compressor to the expander (this is important when the size of the alphabet is large) and the probability distribution is dynamically adjusted using the same adaptation mechanism after each token is processed. The detail of Markov modeling [5,10] and the adaptation mechanism would be described in Section 3.

The blocks not mentioned so far in Fig. 1 are the encoder and decoder blocks. According to the probability distribution provided by the adaptive Markov model, the encoder encodes the input character into compressed data while the decoder decodes the compressed data into the original character. The technique used in the encoder and decoder is arithmetic coding. Since arithmetic coding is not new and source program for small-alphabet arithmetic coding is available in textbook [6] and tutorial paper [9], we would not describe the implementation details of arithmetic coding. However, in Section 4, the idea of arithmetic coding is still described briefly, and the implementation method proposed for large-alphabet arithmetic coding is described in detail. Note that a time-efficient implementation method for small-alphabet arithmetic coding may become very slow when applied directly to large-alphabet arithmetic coding.

3. Adaptive Markov Modeling and Implementation Considerations

Note that the elements of the sequence X in Fig. 1 are processed (encoded or decoded) in serial. When X_i is to be processed, only the elements from X_1 to X_{i-1} have already been processed and are available for estimating the probability distribution for processing X_i (Consider this situation from the expander's view). In general, this probability distribution can be made to depend on all the tokens processed before and their order, i.e. let it be

$$Ps(X_i = v) \equiv P(X_i = v \mid X_1=u_1, X_2=u_2, ..., X_{i-1}=u_{i-1}) ,$$
$$v \in \{1, 2, 3, ..., size\_of\_alphabet\} , \quad (1)$$

where $Ps(X_i = v)$ is the surface probability (coined here) with the underlying condition concealed, and $P(X_i = v \mid X_1=u_1, X_2=u_2, ..., X_{i-1}=u_{i-1})$ is the conditional probability that X_i would have a value of v when the instantiation of $X_1,X_2,...,X_{i-1}$ is $u_1,u_2,...,u_{i-1}$. However, in practice, the value

of the probability, $P(X_i = v \mid X_1=u_1, X_2=u_2, ..., X_{i-1}=u_{i-1})$, cannot be estimated directly because the instantiation of $X_1, X_2, ..., X_i$ just appeared. Therefore, an approximate estimation method must be used.

To estimate the conditional probability practically in equation (1), the technique of Markov modeling is thus adopted. In Markov modeling [5,10], this conditional probability is first assumed to have the Markovian property, i.e. let

$$P(X_i \mid X_1,X_2,...,X_{i-1}) = P(X_i \mid X_{i-1}) , \quad \text{for the first order case,} \tag{2}$$

or

$$P(X_i \mid X_1,X_2,...,X_{i-1}) = P(X_i) , \quad \text{for the zero order case} \tag{3}$$

In equation(2), X_i is assumed to depend on only the last processed token X_{i-1}, but not the other tokens $X_1,X_2,...,X_{i-2}$. In equation (3), X_i is assumed to be independent of all the previous tokens. In addition, in Markov modeling, the conditional probability is assumed to have the stationary property, i.e. let

$$P(X_i =v \mid X_{i-1}=u) = P(X_{i-1}=v \mid X_{i-2} =u) = ... = P(X_2=v \mid X_1=u), \quad \text{for the first order case,} \tag{4}$$

or

$$P(X_i=v) = P(X_{i-1}=v) = ... = P(X_1=v) , \quad \text{for the zero order case} \tag{5}$$

In equation (4) and (5), the probability distribution is not changed with the token index or time. However, for adaptive text compression, the assumption of stationary should not be made because the probability distribution will be adjusted dynamically while the tokens are being compressed. Such re-estimations are needed because initially the probability distribution is set as uniform, which needs to be corrected, and the local characteristics of the input text may be very uneven in different positions of the text.

In this paper, after token X_i is processed, the probability distribution is readjusted in accordance to the following formula,

$$Nuv(i) = Nuv(i-1) + 1 , \quad \text{if} \quad X_{i-1} = u \text{ and } X_i = v ,$$
$$Nuv(i) = Nuv(i-1) , \quad \text{otherwise,} \qquad (Nuv(0) = 0), \tag{6}$$

$$Nu(i) = Nu(i-1) + 1 , \quad \text{if} \quad X_i = u ,$$
$$Nu(i) = Nu(i-1) , \quad \text{otherwise,} \qquad (Nu(0) = 0), \tag{7}$$

$$N(i) = N(i-1) + d , \qquad (N(0) = \text{size\_of\_alphabet}), \tag{8}$$

$$P(X_{i+1} = v \mid X_i = u) = Nuv(i) / Nu(i) , \tag{9}$$

$$P(X_{i+1} = u) = (Nu(i) * d + 1) / N(i) , \tag{10}$$

In the formula above, Nuv(i) counts the number of times that the symbol v is found to immediately follow the symbol u in the instantiation of $X_1,X_2, ...,X_i$. Nu(i) is the occurrence count of the symbol u in the instantiation of $X_1, X_2, ...,X_i$. N(i) represents the total count of all Nu(i) across different u before N(i) is multiplied by d and increased by the initial value. Based on these counts, the probability distribution to process token X_{i+1} is estimated according to equations (9) or (10), depending on the order of the adopted Markov model. In equations (8) and (10), the increment constant d usually is set to be larger than one (e.g. 50) to accelerate the adaptation process and to decrease the effect of the alphabet's size for zero order Markov model. For example, if the alphabet's size is 10,000, the value of d is 1, and the input text is consisted of 10,000 repetitions of a particular symbol u, then according to equation (10) after all the input characters are processed the value of P(u) is only 0.5 .

For Markov model with order one, a serious problem is that Nuv(i) would be zero for most of the (u, v) combinations due to the nature of adaptive processing and the large alphabet size, e.g. there would be 10^8 different combinations if the size of the alphabet is 10,000. This is the called zero-frequency problem [6,11,12,13]. In addition to the zero-frequency problem, the storage of the counts Nuv(i) is another problem that will be discussed in section 4. Because of the zero-frequency problem, equation (9) cannot be used directly for Markov model of order one. Otherwise, those (u, v) combinations with Nuv(i) = 0 would result in infinite entropy [5]. To resolve this problem, a portion of probability called the escape probability must be allocated to those (u, v) pairs not seen so far. To determine how large the escape probability should be set, a few estimation methods [12,13] have been proposed. The method adopted in this paper is proposed by Turing [11,12], which is an approximation to the Poisson process method [13]. In detail, if X_i = u, the escape probability is estimated as

$$\text{Pe} \equiv \text{the escape probability for processing } X_{i+1} = \text{Mu(i)} / \text{Nu(i)}, \quad (11)$$

where Mu(i) is the number of those symbol v with Nuv(i) = 1, i.e. the number of different (u, v) pairs that have occurred only once in the instantiation of $X_1,X_2,...,X_i$. In equation (11), the parameter Mu(i) may be zero or equal to Nu(i-1) in adaptive text compression, which is still a serious problems. Therefore, a slight modification is required, and a typical one proposed by Witten [13] and adopted here is

$$\text{Pe} = (1 + \text{Mu(i)}) / (2 + \text{Nu(i)}), \quad (12)$$

When the escape probability is determined, it can be allocated, to those symbols v not following u in the instantiation of $X_1,X_2,...,X_i$, in proportion to their zero order probabilities $P(X_{i+1}=v)$ defined in equation (10). Now, the more practical formula for the first order probability $P(X_{i+1}=v \mid X_i=u)$ can be given as

$$P(X_{i+1}=v \mid X_i=u) = (1-\text{Pe}) (\text{Nuv(i)} / \text{Nu(i)}), \quad \text{if Nuv(i)} > 0, \quad (13)$$

$$P(X_{i+1}=v \mid X_i=u) = \text{Pe} \cdot (P(X_{i+1}=v) / \text{Pt}), \quad \text{if Nuv(i)}=0, \quad (14)$$

where Pt is the normalization term and is defined as the total probability of those symbols v with Nuv(i) = 0. To save computation time, Pt can be evaluated as

$$Pt = 1 - \Sigma_{\{v \mid Nuv(i) \neq 0\}} P(X_{i+1} = v) \quad (15)$$

4. Data Structure for Fast Arithmetic Coding

First, the idea of arithmetic coding [6,8,9] is briefly reviewed. In arithmetic coding, an input text of any length is always represented by a half-open interval [a, b) , $0 \leq a < b < 1$, as its encoded output. This interval is obtained from the initial interval [0, 1) by a sequence of narrowing operations. Each narrowing operation is performed to encode a token. For example, let $[a_i, b_i)$ be the interval after token X_i is encoded, X_{i+1} be of the value u, and the cumulated probability interval for u be $[c_u(i), e_u(i))$ (so $e_u(i) - c_u(i)$ is u's occurrence probability). Then, the narrowed interval after token X_{i+1} is encoded is defined as

$$[a_{i+1}, b_{i+1}) = [a_i + r_i * c_u(i), \quad a_i + r_i * e_u(i)), \quad (16)$$

where $r_i = b_i - a_i$ is the range of the previous interval. According to this definition, it can be seen that successive tokens when encoded will reduce the range of the interval, and a symbol of greater occurrence probability will have less effect on the reduction than an unlikely symbol. Also, as the index i grows, the interval would become very narrow and the representations of a_i and b_i would have the same bit pattern in the most significant part. In practice, these common bits need not be kept, i.e. the most significant bit would be immediately shifted out and sent to the output each time it is found to have the same value in both the representations for a_i and b_i. Therefore, the complete representations of a_i and b_i have not been kept but the least significant part with different bit patterns.

4.1 IPR Search Tree

According to the formula in equation (16), to encode a token, the cumulative probabilities of this token's instance symbol and its successor symbol are what actually needed. These cumulative probabilities can be computed, when needed, from the frequency counts of the symbols if the concerned symbol group must allow new elements be added dynamically and is small in size, e.g. the group of the symbols v with Nuv(i) > 0 in equation (13). On the other hand, when the size of a concerned symbol group is large, it is better to keep those symbols' frequency counts in the form of cumulated frequency counts. For example, the group of the symbols v, with Nuv(i) = 0 in equation (14), is usually very large in size (more than 10,000 symbols). In this paper, in order to reduce the large memory space required to maintain a separate and large symbol group for each different symbol u of equation (14), an implementation simplification is needed, i.e. only one symbol group consisted of all the alphabet symbols is used as the common symbol group for all different symbol u of equation (14) though some symbol v

may have non-zero Nuv(i) count. Then, the normalization term Pt in equation (14) would have a value of one and can be deleted. Note that, the representation in cumulated frequency counts does not solve the problem completely. After a symbol u is encoded, u's occurrence frequency will be updated according to equation (7) in adaptive text compression, which means that all the cumulated frequencies of the symbols behind u will be updated. To overcome this problem, a type of tree structure called in-path-recording (IPR) search tree is proposed, which is designed independently and is different from the one proposed by Bell, *et al*. [6]. Note that we have provided a self-adjusting mechanism while they didn't.

An example of IPR search trees is shown in Fig. 2. In Fig. 2, each node represents a symbol —

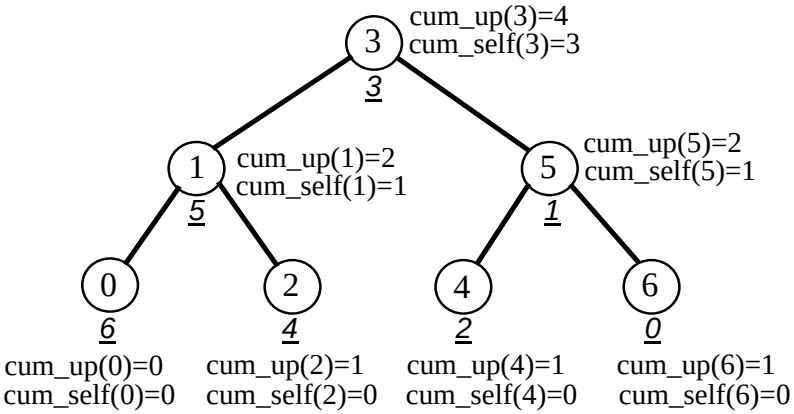

Fig. 2 An example of IPR search trees.

(indicated in the circle) of the alphabet (of six symbols here) except the leftmost one is used to compute the totally cumulated frequency. The cumulated frequencies of the symbols are shown below the nodes and are underlined. And the nodes are adequately linked by the left(k) and right(k) (the left and right son pointers) fields of each node k to form a balanced search tree. In addition, there are two more fields for each node k, i.e. cum_up(k) (used to compute the cumulated frequencies of the nodes in the left subtree), and cum_self(k) (used to compute the cumulated frequency of node k).

For such an IPR search tree to be useful in arithmetic coding, two operations, find_cum(k) and update(k, d), need to be performed efficiently, which are find_cum(k), the function to find the cumulated frequency of symbol k, and update(k, d), the function to increase, by the quantity of d, the cumulated frequencies of all the symbols less than k. The function find_cum(k) can be explained by using the example in Fig. 2. Suppose the cumulated frequency of symbol 1 would be found and saved in a temporary variable w. The value of w is first cleared to zero and the path from the root node (i.e. node 3) to node 1 is traversed. When the left pointer of a node k is followed to traverse this path, the value in the cum_up(k) field is added to w but not added if its right pointer is followed. Finally, when the destination node is reached, i.e. node 1, the value in the field cum_self(1) is added to w. Therefore, the value of w is 5 (cum_up(3) + cum_self(1) = 4 + 1). By using the function find_cum(k), the cumulated probability interval needed in equation (16) can be computed as

$$[c_u(i), e_u(i)) = [\text{find\_cum}(u) / \text{find\_cum}(0), \quad \text{find\_cum}(u-1) / \text{find\_cum}(0)) \qquad (17)$$

Now, the function update(k, d) is illustrated as follows. Suppose the occurrence frequency of node 4 in Fig. 2 is to be increased by 2. The path from the root node to node 4 is then traversed. When a node k's right pointer is followed to traverse this path, both the fields cum_up(k) and cum_self(k) are increased by 2 respectively, but not changed when its left pointer is followed. Thus, cum_up(3) and cum_self(3) are changed to 6 and 5 respectively while cum_up(5) and cum_self(5) remain unchanged. Finally, when the destination node is reached, i.e. node 4, only the field cum_up(4) is increased by 2 but not the field cum_self(4). Thus, cum_up(4) is changed to 3 while cum_self(4) remains. After this update operation is performed, the cumulated frequencies of the nodes, 0 through 6, would be 8, 7, 6, 5, 2, 1, and 0 respectively. These values can be used to verify the functions find_cum(k) and update(k, d). According to the explanation above, it can be seen that these two functions can be performed in logarithmic time if the tree is balanced. Therefore, for a balanced IPR search tree, the time-expensive operation of frequency count updating in large-alphabet arithmetic coding is not a severe problem.

4.2 Self-Adjusting Tree and List

In addition, note that the symbols of the alphabet are not uniformly referenced, i.e. some are very frequently referenced while some are hardly referenced. We can take advantage of this phenomenon to further reduce the running time by dynamically adjusting the structure of the IPR search tree to move the frequently referenced nodes toward the root to reduce its path length. One self-adjusting mechanism that can be adopted is the one used in splay trees [14,15]. However, in this paper, only a simple rotation is performed at the node whose occurrence frequency has just been updated. A simple rotation can be a right or left rotation as shown in Fig. 3. In Fig. 3, the node b, on the left side, is the node to be rotated by a right rotation while the node d, on the right side, is the node to be rotated by a left rotation. |k| represents the occurrence frequency of a node k. The fields Cum_Up(.) and Cum_Self(.) represent the updated versions of cum_up(.) and cum_self(.) for a right rotation, or the original versions for a left rotation. Note that after a simple rotation is performed, only the fields cum_up(.) and cum_self(.) in the node rotated or its parent need to be adjusted to maintain consistency. The required adjustment formula are as listed in Fig. 3. For example, after the occurrence frequency of node b on the left side of Fig. 3 is updated, a right rotation will then be performed to make node b as the father of node d, and the two relevant fields in node b will be adjusted by the formula

$$\text{Cum\_Up(b)} = \text{cum\_up(b)} + \text{cum\_up(d)} \qquad (18)$$

$$\text{Cum\_Self(b)} = \text{cum\_self(b)} + \text{cum\_up(d)}$$

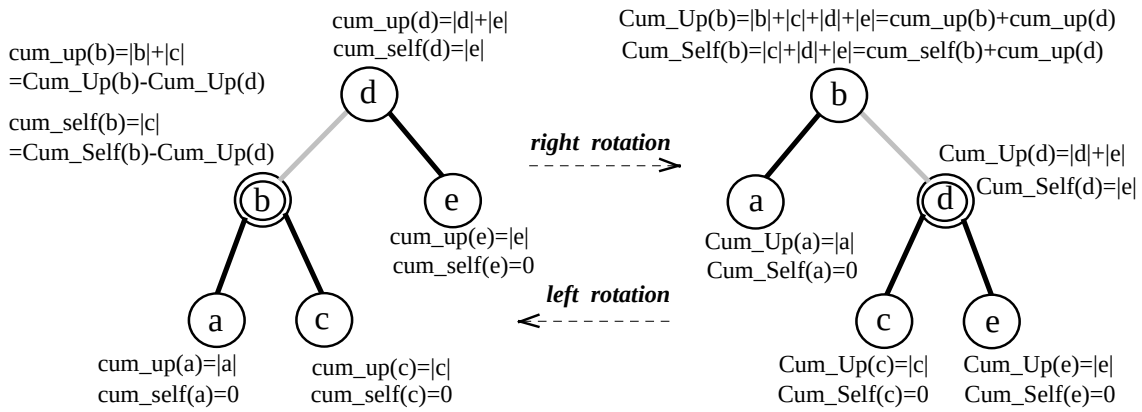

Fig. 3 Example left and right rotations, and the adjusting formula.

The self-adjusting IPR search tree described above is used to compute the cumulated probability of a symbol v with Nuv(i) = 0 in equation (14). To compute the cumulated probability of a symbol v with Nuv(i) > 0 in equation (13), the data structure of self-adjusting list [9,16] is adopted in this paper. That is, for each symbol u, there is a linked list constructed to store those symbols v, found to follow u, and their occurrence frequencies. Here, "self-adjusting" means that the symbol v, when referenced, will be moved to the first position of the list to accelerate the latter references to v. In addition to node reference, strategy about node deletion should also be considered because the main memory is finite and the allocated nodes have to be freed and reused for those newly found (u, v) pairs when the input text is long. To resolve the problem of insufficient memory, the LRU (Least Recently Used) criterion discussed in the "operating system" textbooks is also adopted. In practice, the free nodes and the allocated nodes are linked (by a dedicated field) to form a queue. When a node is referenced, it is move to the rear end of the queue immediately, and gotten from the front end if freed (if already allocated) when it is needed to keep a new symbol v in the linked list of a symbol u.

5. Text Compression Experiments

A series of experiments had been conducted to study the compression efficiency of the approach proposed and the factors that may lead to improvements. This approach is also compared to some popular and commercially developed software packages such as ARJ and PKZIP. Note that all the experiments are conducted physically (not in simulation), i.e. the programs for different processing conditions in our approach are all written as ready-to-use software packages. In these experiments, a 486-33 personal computer is employed as the platform. The main memory available for the programs is about 600K bytes under the ordinary operating mode of the hardware. Borland C++ 3.1 is used as the programming environment. Here, the two Chinese text files used in the experiments are called CX1 and CX2. CX1 is collected from the twelve Chinese textbooks of primary school and its size is 231,737 bytes. CX2 is collect from the editorials of evening newspapers. The editorials were published in a period of about six months, and consisted of 115 articles (each of length between 1K to 2.3K bytes). The size of CX2 is 170,127 bytes.

5.1 Experiments for the Approach Proposed

To study the factors that may reduce the compression ratio or running time in our approach, four conditions are set and tested in the experiments. These four conditions are named as SM0-, LM0-, LM0^, and LM1^. For these names, "S" indicates the limited alphabet consisted of the 256 ASCII characters plus the pseudo character end_of_coding, and "L" indicates the extended alphabet consisted of the 256 ASCII characters, the 13,522 Big-5 characters, and the pseudo character end_of_coding. "M0" represents the zero order Markov model while "M1" represents the first order Markov model. In addition, "-" indicates that only linear lists are used, while "^" indicates that an IPR search tree, instead of a linear list, is used to store the cumulated counts of the counters Nu(.) for different u.

For these four conditions, the corresponding programs had been written and then tested by using the texts CX1 and CX2. The results of these experiments are shown in Table I. In Table I(a), aa

Table I Compression ratios and running time in different conditions

Table I(a) Compression ratios in different conditions

	SM0-	LM0-	LM0^	LM1^
CX1	78.5%	57.0%	57.0%	42.3%
CX2	81.8%	60.1%	60.1%	48.4%

Table I(b) Time spent in different conditions

	SM0-	LM0-	LM0^	LM1^
CX1	14.2s, 24.0s	49.5s, 78.0s	14.3s, 18.6s	35.2s, 33.2s
CX2	10.8s, 18.9s	43.3s, 67.5s	10.5s, 13.7s	25.8s, 24.2s

the compression ratios for different combinations are listed. According to this table, the best compression ratios are 42.1% for CX1 and 48.7% for CX2 in the last column. Therefore, the condition LM1^ (extended-alphabet Markov model of order one and using an IPR search tree) is the best of the four conditions and is suggested to be adopted to obtain the lowest compression ratio. For the difference between 48.7% and 42.1%, note the fact that the themes of the articles in CX2 have more variation than those in CX1 and the model needs to change its statistics more frequently. Another fact is that Chinese characters' occurrence is under control (for teaching purpose) in CX1 while Chinese characters or words are more freely used in CX2. Next, from the first and second columns of Table I(a), it can be found that for both CX1 and CX2, the compression ratios can be lowered (about 20%) when the alphabet is changed from the limited one to the extended one. This indicates that the alphabet should be adequately constructed to let its symbols match the lexical units of the text to be compressed, and that a storage-unit oriented alphabet (though easier to implement) may result in a considerable loss of compression

efficiency. In addition, from the third and last columns of Table I(a), it can be found that for both CX1 and CX2, the compression ratios are reduced further (about 12%) when the first order Markov model is used instead of the zero order Markov model. This shows the existence of correlation between adjacent characters in Chinese text, which can be utilized in text compression and other applications [10].

For processing speed or running time, the results of the experiments are listed in Table I(b). In this table, the number at the left of a cell is the time (in seconds) used by the compressor while the number at the right is the time used by the expander. From this table, it can be found that the time spent would grow rapidly if the alphabet is entended considerably while the data structure of a linear list remains the same. This can be seen from the first and second columns of Table I(b). Therefore, a better data structure should be adopted when the alphabet is large. In the third column of Table I(b), the data structure of an IPR search tree is used instead, and the time spent is reduced drastically when compared to the second column. In the last column, the time spent doesn't grow too much for the more complicated model. This is also due to the use of an IPR search tree. Actually, the programs for the condition, LM1^, can process more than 6K bytes of text per second. In other words, the increased running time are tolerable with respect to the large reduction in compression ratios. In addition, it can be found from Table I(b) that the difference of the time spent by the compressor and expander would be smaller if an IPR search tree is adopted instead of a linear list.

5.2 Comparison with Two Compression Packages

Here, the compression ratios obtained from our approach (in the condition LM1^) are compared with those from the popular software packages ARJ and PKZIP. By using CX1 and CX2 as the example texts, the results obtained from the experiments are listed in Table II. In aaaaa

Table II Compression ratios for different combinations

	LM1^	ARJ	PKZIP
CX1(231,737 bytes)	42.3%	52.8%	53.0%
CX2(170,127 bytes)	48.4%	60.6%	60.8%

Table II, it is obvious that our approach not only obtains the lower compression ratios but reduces these ratios by more than 10% (52.8-42.3=10.5 and 60.6-48.4=12.2). Furthermore, our approach is steadily better than the packages ARJ and PKZIP regardless of the length of the text file to be processed. This can be seen from Fig. 4, which plots the compression ratios against the positions in the text files, CX1 and CX2. The curves for the package PKZIP are not plotted because its aaaa

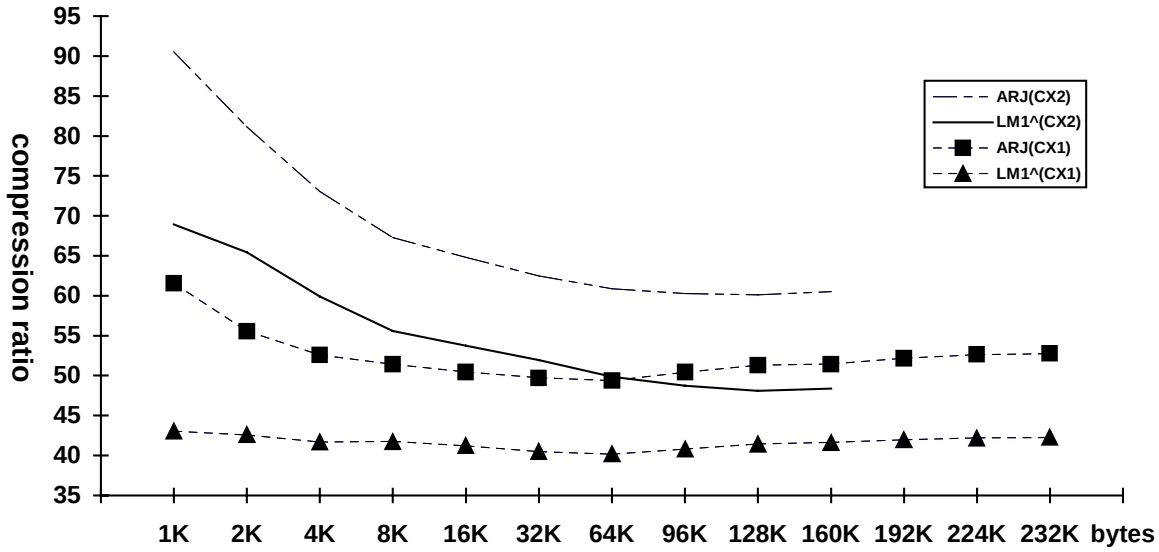

Fig. 4 Compression ratios plotted against positions in the input text.

compression ratios are nearly identical to those obtained from the package ARJ at all positions measured. According to the two pairs of nearly paralleled curves, in Fig. 4, for CX1 and CX2, it is obvious that the compression ratios of our approach are always lower (by about 10%) than those ratios from the packages ARJ and PKZIP. We think the key factor for the better compression efficiency of our approach is the construction of the alphabet. It is certain that the general-purpose compression packages ARJ and PKZIP don't know that Chinese characters are represented in Big-5 code and Big-5 code is a special double-byte code. Therefore, only by extending the alphabet to include the Big-5 Chinese characters, the compression ratios obtained under the condition LM0- and LM0^ in Table I(a) are comparable to those ratios obtained from ARJ and PKZIP.

6. Concluding Remark

In this paper, an approach for Chinese text compression is proposed. By using this approach, compression ratio of less than 50% is achievable. According to the experiments conducted, the ratios obtained are considerably (about 10%) lower than those from the popular data compression packages. The compression efficiency from our approach is due to the adoption of large alphabet to include the large amount of Chinese characters, the use of adaptive first-order Markov model to dynamically track the contextual dependency, and the application of arithmetic coding to encode the data more compactly. In addition, by adopting the LRU criterion, the self-organizing strategy, and the proposed IPR search tree, our approach had been implemented with sufficient efficiency in time and space to make itself practical. If the requirements for time and space efficiencie are relaxed, it is certain that first order Markov model and some other types of models (e.g. a hybrid model using both the statistical and grammatical information) can obtain even higher compression efficiency.

Acknowledgment

Many thanks are to Dr. Ruei-tuen Wu, who provides the text file CX1, and my wife Huei-min Huang who provides the text file CX2.

Reference

[1] C. C. Chang and W. H. Tsai, "A Data Compression Scheme for Chinese-English Characters", Computer Processing of Chinese & Oriental Languages, Vol. 5, pp. 154-182, March 1991.

[2] 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), pp. 205-207, May 1992.

[3] L. Y. Tseng and T. H. Huang, "A Predictive Coding Method for Chinese Text File Compression", Journal of Computer, Vol. 2, pp. 18-23, 1990.

[4] T. C. Bell, I. H. Witten and J. G. Cleary, "Modeling for Text Compression", ACM Comput. Surv., Vol. 21, pp. 557-591, Dec. 1989.

[5] R. W. Hamming, Coding and Information Theory, Prentice-Hall, Inc., 1980.

[6] T. C. Bell, J. G. Cleary and I. H. Witten, Text Compression, Prentice-Hall, Inc., 1990.

[7] D. E. Knuth, "Dynamic Huffman Coding", J. Algorithm, Vol. 6, pp. 163-180, 1985.

[8] G. G. Langdon, "An Introduction to Arithmetic Coding", IBM J. Res. Dev., Vol. 28, pp. 135-149, Mar. 1984.

[9] I. H. Witten, R. M. Neal and J. G. Cleary, "Arithmetic Coding for Data Compression", Commun. ACM, Vol. 30, pp. 520-540, June 1987.

[10] H. Y. Gu, C. Y. Tseng and L. S. Lee, "Markov Modeling of Mandarin Chinese for Decoding the Phonetic Sequence into Chinese Characters", Computer Speech and Language, Vol. 5, pp. 363-377, Aug. 1991.

[11] A. Nadas, "On Turing's Formula for Word Probabilities", IEEE trans. Acoust., Speech, and Signal Processing, Vol. 33, pp. 1414-1416, Dec. 1985.

[12] S. M. Katz, "Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer", IEEE trans. ASSP, Vol. 35, pp. 400-401, March 1987.

[13] 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, pp. 1085-1094, July 1991.

[14] D. W. Jones, "Application of splay tree to data compression", Commun. ACM, Vol. 31, pp. 996-1007, Aug. 1988.

[15] M. A. Weiss, Data Structures and Algorithm Analysis in C, the Benjamin/Cummings Publishing Company, Inc., 1993.

[16] J. H. Hester and D. S. Hirschbersg, "Self-organizing linear search", ACM Comput. Surv., Vol. 17, pp. 295-311, Sept. 1985.