Mathematical Problems of Computer Science 17, 1997, 52-56.

Practical Implementation of the Lossless Compression Algorithm

G. H. Khachatrian and A. B. Andreasyan

Institute for Informatics and Automation Problems
of NAS of RA and YSU

In this paper a combination of the LZ78 method with a new scheme of model contexting is introduced. In the proposed scheme the hashing function is also used. This approach speeds up the searching process and has an improvement over model contexting.

1 Introduction

Most of the existing compression applications are based on the schemes proposed by Ziv and Lempel [1], [2]. In [1] (denoted to be a LZ77) it was proposed to replace each matching by the $(C_{i1}C_{i2}C_{i3})$ where C_{i1} is the pointer to the start position of matching , which has the length $l(C_{i1}) = \log(n - L_s)\rfloor$, where L_s is the length of a lookahead window, n is the size of the buffer, C_{i2} is the length of matching and C_{i3} is the first unmatched symbol in the lookahead window.

In practice the algorithm described in [2] (denoted to be a LZ78) is often used. LZ78 starts encoding with empty dictionary (buffer). As the encoding continues , fairly long matchings are collected in the buffer and each matching is replaced by the pointer and by the length of the matching (it will be denoted by the pair (p,l)). Testing shows that this method is more effective in compression scheme than LZ77.

There are some schemes for coding of the symbols by the context modelling. These schemes use some previous symbols of the text to predict the next symbol. This approach allows to obtain good compression ratio for the textual files. Disadvantages of these schemes are the large memory requirement and the low speed. These schemes allow on-line encoding too.

In this paper a combination of the LZ78 method with a new scheme of model contexting is introduced. In the proposed scheme the hashing function is also used. The hash value of the previous symbols is computed and depending on that value, the current symbol in the corresponding part of the memory (denoted to be a hash box) is searched. This approach speeds up the searching process. If the current symbol is found, it is coded by the dynamic Huffman tree, otherwise it is added to the symbol chain.

2 Description of the algorithm

A proposed compression algorithm is based on LZ78 approach and model contexting. We use a local dictionary (ring buffer of size N+F, where N is the size of buffer and F is the size of sliding window) and let M is the size of memory (which will be denoted by the term "model") for model contexting. Note, that $N=2^k$, where k depends on the resource of the coder. Proposed on-line encoding and decoding algorithms have the following form:

2.1 Encoding Algorithm

a. Get the current matching (p, l) by LZ78.

Check the corresponding hash box. If the box is not empty and l > threshold encoder output is "0"(if the hash box is empty bit is not set) and at the beginning l is coded by the adaptive Huffman tree ,then p is coded by the method described in 1.2. Otherwise (l< threshold) only a current symbol have been coded.

If the current symbol exists in corresponding hash box , encoder output is "1" and the current symbol is coded through dynamic1 Huffman tree. If not, encoder output is "0" and the current symbol is coded by the adaptive Huffman tree.

b. Update the local dictionary and the corresponding hash box.

2.2 Decoding Algorithm

1. Initialise the local dictionary and model by performing step 1 of the encoding algorithm.
2. **repeat forever**

a. *If decoder reads "1" then the current stream of binary bits is decoded by the hash box through dynamic Huffman tree. If decoder reads " 0" then the current stream of binary bits is decoded by the adaptive Huffman tree.*

If the decoding value is greater than 256 it means that l has been encoded and follows p. Otherwise the current symbol is encoded.

b. *Update the local dictionary and the corresponding hash box by performing step 2.b of the* ***encoding*** *algorithm.*

Note, that when the compression ratio is more important than the speed then the second searching is implemented (lazy evaluation). The current symbol enters into the local dictionary and the search is continued with one shift. If longer matching is found, the previous symbol is coded by the adaptive Huffman tree or by the model, and current long matching is coded. If not, the previous matching is coded. Updating of the local dictionary is done by the identity heuristic[3].

We perform the coding by the following way. Let (l,p) be the current matching where p is a pointer to the start position of the matching and l is the length of matching and let n be the pointer to the current updated position in the local dictionary. First l is coded by the adaptive Huffman tree. Pointer to the start position is coded by the following way

$$\frac{(p-n) \bmod (N-1)}{2^i} = k * 2^i + r \quad i=1,q \qquad (1)$$

where i is defined by N. $\frac{N}{2^i}$ corresponds to the nodes of Huffman tree, assigned for coding of k. To speed up the search in process N is choosen to be equal to 2^j, where j>q. It follows from (1), that all matchings, for which $k*2^i < p - n < k*2^{i+1}$, have the same Huffman code. So the pair (l,p) is replaced by the code $(C_1(l),C_2(k),r)$, where $C_1(l)$ is a regular Huffman code of l, $C_2(k)$ is a Huffman code for k.

3 Fast parsing of string by the hash function

For the fast parsing a hash function is used. As the arguments of the hash function symbols from

the file are taken beginning from the current position.As a hash function of (a,b,c), the following transform is used

$$(a*2^3 + b*2^8 + c) \bmod 8191 \quad (2)$$

where a, b, c are the symbols of the file. Each hash value can correspond to one or more positions in a local dictionary. These dictionary positions are ordered by ascending.

In order to parse the string, the hash value is computed and if this value is found in the hash table, then the symbols from the corresponding position of the dictionary are compared with the symbols starting at the current position. So we can find some strings of different length, from which we choose the longest and the nearest one. After coding of the current string , the hash table is updated. When the local dictionary is filled hash values are replaced by the new values.

The proposed model has the following construction. Let M be the size of memory

4 Model contexting

In the case, when matching length is shorter than the threshold value model contexting is used. A well-known k-th order Markov model can be represented by the following way : the 0 order (level) is intended for all 256 ASCII characters , the 1-st order is intended for all the characters following 0 order and so on ... .The symbol belongs to the k-th level if the previous symbol belongs to the (k -1)-th level. In order to the use of the k-th order model the frequency occurrences of the k-th level symbols must be kept. This approach gives a good compression for textual files, but too much memory and a long time are required to find the current symbol. for model contexting. The first section of the memory consists of 8191 bytes corresponding to the modulo of hash function. This section of memory is divided into $k=(8191+1)/2^5 = 2^8$ (2^5 is obtained from the experimentally) parts. The size of each part is equal to 256 bytes, which corresponds to the ASCII characters . Other section of the memory M - 8191 - 1 is divided into 2^8 zones. The size of each zone is equal to $(M-8192)/2^8$, where the frequency of occurrences of the current symbol and pointer to the next symbol, which have the same hash value and follow the same symbol are stored. The set of this symbols will be refereed as the hash box. Each starting part corresponds to the appropriate zone of the memory.

When l < threshold (for this case threshold=5,which is obtained by the test) and the current symbol exists in the hash box, then it will be coded by the dynamic Huffman tree code. Otherwise (when current symbol is absent in the hash box) it is coded by the adaptive Huffman tree code. First, for the current symbol the hash value is computed and divided by 2^8

$$h = ((a* 2^3 + b* 2^8 + c) \bmod 8191) / 256 \quad (3)$$

where a,b,c are the previous three symbols. Then in the corresponding starting part, it is checked if the symbol "c" has the hash box. If yes, then the current symbol is searched in the hash box. If current symbol is not found then it is added to the hash box. When the corresponding zone is filled ,the content of the hash box is rotated, i.e. the first symbol is replaced by the last one.

To be decoded correctly encoder must use overhead bits. "0" overhead means that the current symbol is absent in the hash box or hash box is not empty. "1" overhead means that current symbol is found in hash box.

The experimental results show, that by this method the obtained average compression ratio is of 0.4375, i.e. each ASCII character is coded by the 3.5 bits in average. The proposed model can be applied to the binary sources too.

5 Compression for "executable" files

The performance of any data compression method highly depends on the data being compressed. As a sample three kinds of files are considered, namely textual source, binary source and special kind of binary source - the "executable" file.

For the last type of the files one may apply a small memory size. The following approach is implemented. Except the local dictionary, where the matchings are stored, two additional dictionaries are kept too. In the first dictionary (denoted to be a position buffer) the current number of each matching is kept, the length of which is greater than some threshold, taken to be equal to 3. In the second dictionary (denoted to be a length buffer) the length of this matching is stored. A new symbol in adaptive Huffman tree table is introduced. Let it has the number $256 + F + 1$.

The **2.1 Encoding Algorithm** and **2.2 Decoding Algorithm** will be in the following form:

1. Initialise the local, position and the length dictionaries.

2. **repeat forever**

a. Get the current matching (l,p).

If l > threshold, then p is searched in the position dictionary and l is searched in the length dictionary.

If p and l are found, then symbol(256 +F +1) is coded by the adaptive Huffman tree code and the current number is coded by the prefix code.

If not, then the (l,p) is coded by method described in the section[1].

b. Update the local, position and the length dictionaries.

1. Initialise the local, position and the length dictionaries by performing step 1 of the encoding algorithm.

2. **repeat forever**

a. Decode the current stream of binary bits.

If 256 +F +1 is obtained, then the (l,p) pair is obtained from the length and the position dictionary, otherwise by the method described in the section[1].

b. Update the local, position and the length dictionary performing step 2.b of the encoding algorithm.

This method is effective for the executable binary files.

This method allows good compression ratio having small buffer.

6 Experimental results

Table 4.1 shows compression results

*.txt	*.doc	Dictionary size (bytes)	*.exe
39823 0.2512	48655 0.2997	1024	37302 0.4634
38706 0.2442	43902 0.2704	2048	36747 0.4563
37630 0.2374	40119 0.2471	4096	36435 0.4525
36961 0.2332	37674 0.2321	8192	

36515 0.2304	35632 0.2195	16384	
36283 0.2289	32918 0.2028	32768	

For the comparison the compression ratio obtained by the pkzip.exe 2.04 and arj.exe 2.30 are given.

pkzip.exe 2.04	36944 0.2331	32852 0.2024	36630 0.4549
arj.exe 2.30	36594 0.2309	33976 0.2093	35967 0.4467

References

1. J.Ziv and A.Lempel "A Universal Algorithm for Sequential Data Compression", EEE Transactions on Information Theory, vol.IT-23,pp 337-343,May 1977
2. J.Ziv and A.Lempel " Compression of individual sequences via variable-rate coding", IEEE Transactions on Storer " Data compression methods and theory", Computer Science press, 1988
3. Theory, vol. IT-24,pp.530-536,Sept. 1978
4. Storer " Data compression methods and theory", Computer Science press, 1988

1. Storer " Data compression methods and theory", Computer Science press, 1988
2. Theory, vol. IT-24,pp.530-536,Sept. 1978
3. Storer " Data compression methods and theory", Computer Science press, 1988