Eleventh International Workshop on Algebraic and Combinatorial Coding Theory
June 16-22, 2008, Pamporovo, Bulgaria *pp. 221-228*

Procedures of extending the alphabet for the PPM algorithm

Radu Radescu `rradescu@gmail.com`
George Liculescu
Polytechnic University of Bucharest, ROMANIA

Abstract. In this paper it is presented the lossless PPM (Prediction by Partial string Matching) algorithm and it is studied the way the alphabet can be extended for the PPM encoding so it will allow the use of symbols which are not present in the alphabet at the beginning of the encoding phase. The extended alphabet can contain symbols with the size larger than a byte. The paper presents the manner to extend the alphabet and the changes that need to be made to the PPM algorithm in order to use the extended alphabet. Three ways to extend the alphabet are proposed: manual, through a run over the text (executed before the encoding phase), and specialized (adapted during the evolution of the algorithm).

1 Introduction

Let us presume that a file contains a string of bytes (characters), which appears many times in the file. PPM must encode independently every byte from the string with a probability (which is preferable to have large value). Every time the character was not found in the past (the string preceding the current context), an escape symbol is send to decrease the level, leading to increment the information from the compressed data flow. The alphabet used by the PPM algorithm has 256 characters (all the characters that can be formed using 8 bits). If the regular alphabet is extended adding a new symbol (the string mentioned above) the algorithm could perform a good compression. An extended alphabet is an enriched known alphabet with a series of symbols that will not be presented in the alphabet offered to the decoder. The symbols that extend the alphabet need to be obtained in the decoding phase through different methods, so while decoding the alphabet will be enriched with new symbols. In the coding phase, the symbols that will extend the alphabet are known, but at the decoding these will be deduced gradually. Three solutions to extend the alphabet are considered:

1. manual adding of the words by the user;
2. search of the words that get repeated using a certain criterion (length, number of appearances, etc.) in a first step by running through the entire text and then adding these words to the alphabet;
3. adaptive search of the text words during the algorithm evolution.

In this paper, we consider that internal words are present at the decoding phase because they are internally generated, and they can be reproduced at the decoding.

2 The manual adding of text words

From the three options, this one is the simplest because it is based on the user experience and has no need for additional processing. For adding of a text word from the manual point of view, it is necessary to define all the parameters involved: length, number of appearances, presence at the decoding, and gain. The length of the word to be added is known. The number of appearances and the presence at the decoding must be set up manually. The set up number of appearances and the length will determine the gain of the text word. In this way, we can set manually the significance of the word. Implicitly, the manual added words would be marked as external.

3 The search of words by running through the text

The suggested solution in this paper is based on the suffix vector. This contains all the suffixes from a string, lexicographically ordered. For example, if we consider the abracadabra string, then the suffixes of this string will be (see Figure 1):

Suffix	Position
abracadabra	0
bracadabra	1
racadabra	2
acadabra	3
cadabra	4
adabra	5
dabra	6
abra	7
bra	8
ra	9
a	10

sorting $\Longrightarrow$

Suffix	Position
a	10
abra	7
abracadabra	0
acadabra	3
adabra	5
bra	8
bracadabra	1
cadabra	4
dabra	6
ra	9
racadabra	2

Fig. 1. The suffix vector.

To obtain a suffix vector we need to extract the suffixes and sort them. It can be noticed that in the suffix vector the side elements can have identical characters. We aim only the strings that start with the first character from the left of the suffix and are continued through the right. Based on this we can determine strings that are repeated in the text. These characters can make up words, which can be used to extend the alphabet used at the encoding In the suffix vector, we can find many words that are repeated but only few of them will be of some interest. To find out which words are significant we will need to induce some restrictions. To do this we will attach to every word a gain with which we will determine its significance. The restrictions will be related to the minimum length, the maximum length, and the minimum gain.

If a match does not have at least the minimum length then it is treated like it did not exist. In order to calculate the gain we need to know the length and the number of appearances. We know the length of the word but finding the number of appearances is a problem. To solve this problem we need to have a view only of words that do not overlap. The words that overlap cannot be compared because an exact delimitation does not appear so we cannot say that both of them exist at the same time in the compression string, because only one word can be coded. Knowing the text from where the suffixes are extracted, it is enough to keep a position vector from where the suffixes begin and a length vector, which will indicate the accepted lengths for every suffix, which begins at that position. When we want to sort the suffixes, we compare the suffixes that start at the specified positions in the position vector. The sorting of the suffixes will result in an arrangement of the positions in the position vector and the of length in the length vector (see Figure 2). The length vector, as it will be shown next, will help to solve the overlapping of words extracted from the text. The length of a suffix can be at most the maximum set up length. To keep track of the found words we will create a list, where every record will be made up of the positions where the word is found and the length of the word. The number of positions on which the word is situated will give the number of appearances.

Position	Length
0	l_0
1	l_1
2	l_2
3	l_3
4	l_4
5	l_5
6	l_6
7	l_7
8	l_8
9	l_9
10	l_{10}

sorting $\Longrightarrow$

Position	Length
10	l_{10}
7	l_7
0	l_0
3	l_3
5	l_5
8	l_8
1	l_1
4	l_4
6	l_6
9	l_9
2	l_2

Fig. 2. The sorting of suffixes.

A record from the list will look like this (see Figure 3):

Position 1	Position 2		Position n	Length

Fig. 3. A recording from the word list.

Example. We will run through the suffix vector and we will compare, in this order, pairs of suffixes: a with *abra, abra* with *abracadabra*, $\cdots$, *dabra* with ra, and ra with *racadabra*. For every pair of suffixes, we will find identical characters or not. Every time we find a word, this will be added in the list by creating a new record (in the case the word does not exist in the list) or by adding the position where it was found. We can see that *a* and *abra* have only the a character in common, and *abra* and *abracadabra* have the *abra* string in common. We can say that *a* is found in the *a* & *abra* suffixes, and in *abracadabra.* So the a recording will have three positions. *abra* is found in *abra* and in *abracadabra* so the *abra* recording will have two positions. We can see that the positions from *abra* can be found in *a*, because *a* is part of *abra.* This means the shorter strings from the word list that are part of other longer strings must have at least the positions of the longer strings.

Remark. If two suffixes do not have a character in common, then from that point there will be no suffixes having characters in common with the suffixes before that point. This means that portions of the suffix vector can be treated separately. For example, the suffixes from below do not have any character in common with the rest of the suffixes (see Figure 4).

Suffix	Position
a	10
abra	7
abracadabra	0
acadabra	3
adabra	5

Fig. 4. Suffixes that have at least one character in common.

Because we mentioned above that shorter strings that are part of longer strings must have at least the positions of the longer strings, we need to find a practical way to accomplish this. We can see that if between two suffixes on the p1 and p2 positions there is a n length match, then we need to update all the recordings smaller or equal to n and that are involved in that match. In order to reduce the number of steps, every time a match is found, and is smaller than the last record from the list, all the element from the list smaller or equal to n will be moved, and we will store the pmoved position in the list in which the element has been moved. This means that pmoved will be the end of the list before the move. From the pmoved position to the end of the list, the p2 position will be added in every record. If the n length match does not exist in the list, it will be added.

Example. In Figure 4, a and abra have in common a so a will be added in the list together with positions 10 and 7 because it is a new word and the list is empty. abra and abracadabra have abra in common so it will be added the 0 position to the previous recordings, after that abra is added in the list with 7 and 0 positions because it is a new word. This means that until now the list contains a and abra. abracadabra and acadabra have in common only a. Because a has length 1, it is smaller than the last recording (abra) so any word with length smaller or equal to 1 will be moved to the end of the list. As a result, the list that contained the recordings (in this order) a, abra now will have the recordings abra, a. This means that the last record will be added to the position where the new a was found, and this is 3 (see Figure 5).

a	10, 7

$\Longrightarrow$

a	10, 7, 0
abra	7,0

$\Longrightarrow$

abra	7, 0
a	10, 7, 0, 3

Fig. 5. Suffixes that have at least one character in common.

Because we need all the words from the list that have at least one character in common with analyzed suffixes, we cannot solve the overlapping between the words until there will be no suffixes that would have a common character with the ones in the list. This way, we will solve the overlapping each time there are no characters in common between the two currently analyzed words. The solving of the overlapping will be handled only inside of a recording from the list, because it is very demanding the check the positions of every recording through the comparison with the other recordings. In addition, keeping of some positions must be settled according to the gain. If there is an overlapping of the recordings, we cannot calculate the gain. This means that the adopted way will be to search the word that has a maximum gain and its positions do not overlap one another but can overlap with other words. After finding this word, we will mark the positions from the input text where the word is found as being occupied. After this, we repeat the search algorithm for a new word. In this manner, the founded words will not overlap. This procedure will repeat until no word can be found under these conditions.'

4 Adaptive search of the words

The first two methods can be used along with any compression algorithms. Only the PPM algorithm uses the method described below. To form the words, we will use the tree, with the help of which the contexts are maintained. The tree is changed every time we wish to insert a word. The tree has all the past contexts, if it has not been emptied to save memory, or only a part of them (from a near past). In the case the inserted word has been preceded by the same context in the past, the number of appearances is incremented and added to the actualization list. In the case the PPM context has been spotted as being followed by the algorithm word 5 times, we can say that in the past PPM algorithm has been seen 5 times. This means a word discovery can be made similar to the run through method previously described. Therefore if a minimum length, a maximum length and a minimum gain are set, some words created with the tree can be considered.

When a word, which is added in the three, has been seen preceded by a certain context, the tree is run through, starting from that word preceded by the context to the root. This means we begin at the node that has the certain word and we move to his parent, then his grandparent, and so on until we reach the root. Every time a move is made, the word from the current node is inserted in a stack and a *total_length* variable is incremented that holds the length of the word added in the stack. If the value of *total_length* is at least the minimum set length then the word preceded by that context is considered for further checking. In the case the gain calculated based on the minimum of *total_length* and the maximum set length and based on the number of appearances of the word preceded by the context is at least the minimum set gain, then the word preceded by the context is considered for further checking. Next, the words are extracted one by one from the stack and are concatenated until the stack is empty or until the maximum length is reached. This way, a word is constructed that may of may not be added to the alphabet. If the word is not in the alphabet, then it is inserted. The added word will have a number of appearances equal to zero and will be marked as being intern so it will be present at decoding.

Normally we are tempted to be less restrictive with the limits imposed on making of a new word, for it to exist in the alphabet. A word that already exists in the alphabet has the chance to be used as soon as possible. If the limits are too restrictive, the process of using a word will be much delayed. The setting of less restrictive rules will produce the negative effect of congesting the alphabet because numerous words are in the set limits. Because of this, we must have a compromise. Because the less restrictive rules allow the words to be used soon, the problem of the large number of generated words must be solved. The solution is the periodical cleaning of the alphabet (after a number of bytes). This way, we will search words that have not been used and are marked as being intern. The words are unused if the real appearances number (set from inside) is zero. If a word has been used at least one time in the encoding stage then it remains in the alphabet, and can no longer be eliminated. This may lead to the growth of the alphabet. This is why a maximum admitted memory would be set for the storing of words. If this memory is exceeded, then the extern words are kept and the intern ones are sorted decreasingly by the gain. Another memory limit will be set for the reducing of the alphabet. The structure is presented in Figure 6.

Example. In Figure 6, we considered that the last added word, a, had the length 1. Therefore, the actualization list will be formed from the nodes with thick lines. We consider the restrictions: *minimum_length = 3, maximum_length = 6*, and *minimum_gain = 6*, the gain being computed using the formula *length* $\times$ *length* $\times$ *appearances*. We can form 3 words at most because there are 3 thickened nodes besides the root, which cannot participate at the forming of the word because they do not contain a word. The 3 words will be:

1) The first node that is checked is the thickened one from the level 3. We insert in a stack all the words from the path that starts at the current analyzed node and ends at root. In order to do this, we use pointers that indicate the parent of every node. In the stack, the words will be placed in the following order: a, a, merge. The total length is 6. We can see that the *total_length*, which is 7, is larger than the minimum length, which is 3. *mergeaa* was seen 7 times.

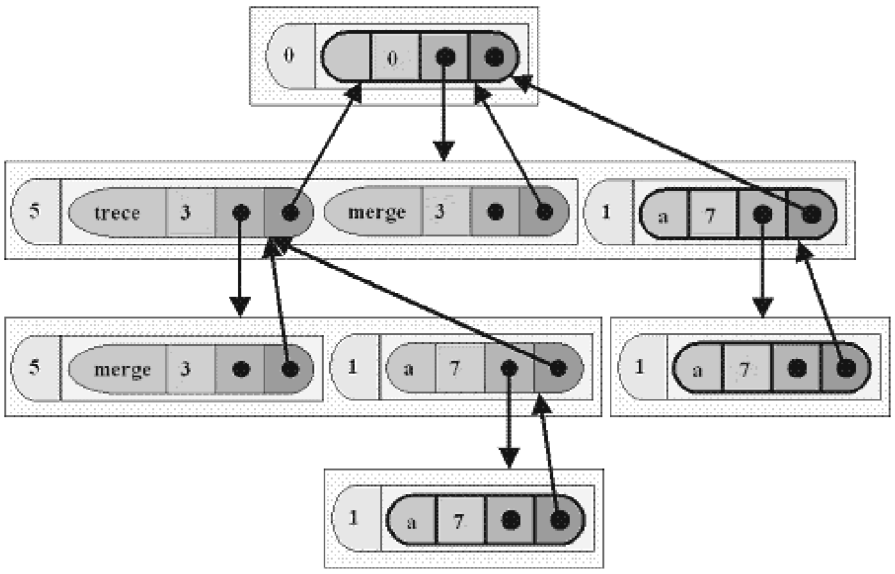

Fig. 6. The adding of the pointer that indicates the parent.

As a result, the gain will be *length* $\times$ *length* $\times$ *appearances*= $6 \times 6 \times 7 = 252$, because the minimum between the total length and the maximum length is 6. We can observe that this gain is bigger than the minimum gain, which is 6. This means this word has all the characteristics to be used to extend the alphabet. Next, we extract the elements from the stack and we concatenate them until we reach the maximum length or the stack is left with no elements. The result string will be: *mergea* (was going in Romanian) and not *mergeaa*, because the maximum length is 6 (see Figure 7). The word is checked if it already exists in the alphabet. If it is not present, then it is added.

$$\begin{array}{|c|}\hline \text{merge} \\ \hline \text{a} \\ \hline \text{a} \\ \hline \end{array} \Longrightarrow \begin{array}{|c|c|c|}\hline \text{merge} & \text{a} & \text{a} \\ \hline \end{array}$$

Fig. 7. The creation of a text word based on the information from the tree.

2) The second node that will be checked is the thickened one from level 2. We insert the words in the stack in the following order: a, a. The total length is 2 and it is smaller than the minimum set length, which is 3. This means we cannot form a word because it does not match the set limits.

3) The third node that will be checked is the thickened one from level 1. This time the only word inserted in the stack is a. Because the total length is 1, we will not be able to create a word with the use of this node, because the minimum length is 3. It is not possible to create a new word with only a node from the first level, even if this matches the set limits, because the word from a single node is certainly in the alphabet.

References

[1] A. Moffat, Implementing the PPM Data Compression Scheme, *IEEE Trans. Commun.* 38, 1990.

[2] Th. H. Cormen, *Introduction to Algorithms*, Ch. E. Leiserson , R. L. Rivest eds., The MIT Press. 1999.

[3] N. Abramson, *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[4] T. Bell , I. H. Witten, J. G. Cleary, *Modeling for Text Compression*, ACM Computing Surveys 21, 1989.

[5] Pr. Skibinski, PPM with the Extended Alphabet, *Inform. Sci.* 176, 2006.

[6] J. Bentley, D. McIlroy, Data compression using long common strings, *Inform. Sci.* 135, Part 1-2, June 2001.

[7] A. T. Murgan, *The Principles of Information Theory in Information and Communication Engineering*, Romanian Academy Press, Bucharest, 1998.

[8] R. Radescu, *Lossless Compression - Methods and Applications*, Matrix Rom Press, Bucharest, 2003.

[9] M. Nelson, *The Data Compression Book*, 2nd Edition, Jean-Loup Gailly ed., M&T Books, 1995.

[10] *** *Data Compression - The Complete Reference*, 3rd Edition, David Salomon ed., Springer-Verlag, 2004.

[11] *** *Lossless Compression Handbook*, 1st Edition, Khalid Sayood ed., Academic Press, 2002.

[12] R. Radescu, C. Harbatovschi, Compression methods using prediction by partial matching, *Proc. 6th Intern. Conf. Commun.*, Bucharest, Romania, 2006, 65-68.

[13] R. Radescu, R. Popa, On the performances of symbol ranking text compression method, *Sci. Bull. "Politehnica" Univ. Timisoara, Romania, Trans. Electr. Commun., special issue dedic. Electr. Telecomm. Symp.* 49, ETC 2004, 25-27.

[14] The Calgary Corpus: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

[15] www.winrar.com