Mapping Words into Codewords on PPM*

Joaquín Adiego and Pablo de la Fuente

Depto. de Informática, Universidad de Valladolid, Valladolid, Spain
{jadiego, pfuente}@infor.uva.es

Abstract. We describe a simple and efficient scheme which allows words to be managed in PPM modelling when a natural language text file is being compressed. The main idea for managing words is to assign them codes to make them easier to manipulate. A general technique is used to obtain this objective: a dictionary mapping on PPM modelling. In order to test our idea, we are implementing three prototypes: one implements the basic dictionary mapping on PPM, another implements the dictionary mapping with the separate alphabets model and the last one implements the dictionary with the spaceless words model. This technique can be applied directly or it can be combined with some word compression model. The results for files of 1 Mb. and over are better than those achieved by the character PPM which was taken as a base. The comparison between different prototypes shows that the best option is to use a word based PPM in conjunction with the spaceless word concept.

Keywords: Text Compression, PPM, Dictionary Algorithms, Natural Language Processing.

1 Introduction

In modern computational environments, processing times and storage costs have been reduced. On the other hand, the amount of data stored and transmitted has increased dramatically. Although most data is multimedia, the amount of textual data, predominant a few years ago, is not negligible. Information Retrieval Systems and Digital Libraries are systems where textual information, with and without format, is still predominant. Besides, these systems are used in several environments such as networks, optical and magnetical media. In these cases, the use of compression techniques is the best choice to solve storage problems and improve access time in storing and processing. Improvements in processing times are achieved thanks to the reduced disk transfer times necessary to access the text in compressed form. Since processor speeds in the last few decades have increased much faster than disk transfer speeds, trading disk transfer times for processor decompression times has become a much better choice [21]. On the other hand, the use of compression techniques reduces transmission times

* This work was partially supported by the TIC2003-09268 project from MCyT, Spain.

and increases the efficiency using communication channels. These compression propierties allow us to keep costs down.

Classical text compression algorithms perform compression at the character level. When an algorithm is adaptive then the algorithm slowly learns correlations between sequences of characters. However, the algorithm usually has a chance to take advantage of longer sequences before either the end of input is reached or the tables maintained by the algorithm reach their capacity. If text compression algorithms were to use larger units than single characters as the basic storage element, they would be able to take advantage of the longer range sequences and, perhaps, achieve better compression performance. Faster compression may also be possible by working with larger units [13].

In this paper, we explore the use of a word representation as the basic unit in PPM, one of the most promising lossless discrete-data compression algorithms at the character level, which uses Markov models of order k.

When the source file is a natural language document, we have no difficulty in recognizing a word as consisting of a sequence of consecutive letters. Each word is separated from the next by space and/or punctuation characters. Following the same approach as Bentley et al. [6], we generalize slightly by considering a natural language text file to consist of alternating alphanumeric-strings and punctuation-strings, where a word-string is a maximal sequence of alphanumeric characters and a punctuation-string is a maximal sequence of non-alphanumeric characters. We use the generic name *word* to refer to either an alphanumeric string or a punctuation string. The generalization allows us to decompose all kinds of text files into sequences of words. We should be able to take advantage of the fact the the alphanumeric and non-alphanumeric words strictly alternate.

The following sections of this paper will consider the problem of generalizing a PPM based compression algorithm to be word-based, then particular PPM word-based algorithms will be described, and finally some experimental results will be reported. Finally, our conclusions and future work are presented.

2 Natural Language Text Compression

With regard to compressing natural language texts the most successful techniques are based on models where the text words are taken as the source symbols [15], as opposed to the traditional models where the characters are the source symbols.

In an English text, for example, words follow a Zipf law, that is, the relative frequency of the i-th most frequent word is $1/i^\theta$, for some $1 < \theta < 2$ [20,3]. On the other hand, the model size (assigning a codeword to each different text word) is not significant in large text collections. Heaps law establishes that the number of different words in a text of n words is $O(n^\beta)$ for some β between 0.4 and 0.6 [12,3]. Thus, the model size grows sublinearly with the collection size.

Natural language is not only made up of words. There are also punctuation, separator, and other special characters. The sequence of characters between every pair of consecutive words is called a *separator*. Separators must also be considered to be symbols of the source alphabet. There are even fewer different separators than different words, and their distribution is even more skewed. Note that, since words and separators strictly alternate in the text, we can have two separate source alphabets, usually leading to better compression. As explained in the Introduction, we will use the generic name *words* to refer to both text words and separators in this paper.

Words reflect much better than characters the true entropy of the text [4]. For example, a semiadaptive Huffman coder over the model that considers characters as symbols typically obtains a compressed file whose size is around 60% of the original size, in natural language. A Huffman coder, when words are the symbols, obtains 25% [21]. Existing compression algorithms that consider the input as a sequence of words are ad hoc in nature. The scheme described by Bentley et al. [6] maintains a list of words sorted into least-recently used order. A word is encoded by its position in this dynamically changing list. Words near the front of the list tend to have shorter codes than those near the end and, assuming words in frequent use stay near the front of the list, compression is achieved. Another example is the WLZW algorithm, which uses Ziv-Lempel on words [10].

Since the text is not only composed of words but also separators, a model must also be chosen for them. An obvious possibility is to consider the different inter-word separators as symbols too, and make a unique alphabet for words and separators. However, this idea is not using a fundamental *alternation* property: words and separators always follow one another. In [15,5] two different alphabets are used: one for words and one for separators. Once it is known that the text starts with word or separator, there is no confusion on which alphabet to use. This model is called **separate alphabets**.

In [17,21] a new idea to use the two alphabets is proposed, called **spaceless words**. An important fact that is not used in the method of separate alphabets is that a word is followed by a single space in most cases. In general, it is possible to be emphasized that at least 70% of separators in text are single space [15]. Then, the spaceless words model take a single space as a *default*. That is, if a word is followed by a single space, we just encode the word. If not, we encode the word and then the separator. At decoding time, we decode a word and assume that a space follows, except if the next symbol corresponds to a separator. Of course the alternation property does not hold anymore, so we have a single alphabet for words and separators (single space excluded). This variation achieves slightly better compression ratios in reported experiments.

3 *k*-th Order Models

These models assign a probability to each source symbol as a function of the *context* of *k* source symbols that precede it. They are used to build very effective compressors such as Prediction by Partial Matching (PPM) and those based on the Burrows-Wheeler Transform (BWT).

PPM [9,18] is a statistical compressor that models the character frequencies according to the *context* given by the *k* characters preceding it in the text (this is called a k-th order model), as opposed to Huffman that does not consider the preceding characters. Moreover, PPM is adaptive, so the statistics are updated as the compression progresses. The larger k is, the more accurate the statistical model and the better the compression are, but more memory and time is necessary to compress and uncompress.

The PPM technique can be viewed as blending together several fixed-order models to predict the next character in the input sequence. More exactly, PPM uses $k+1$ models, of order 0 to k, in parallel. It usually compresses using the k-th order model, unless the character to compress has never been seen in that model. In this case, it switches to a lower-order model until the character is found. The coding of each character is done with an arithmetic compressor, according to the computed statistics at that point. Well known representatives of this family are Shkarin/Cheney's *ppmdi* and Bloom/Tarhio's *ppmz*.

The BWT [7] is a reversible permutation of the text that puts together characters having the same k-th order context (for any k). The BWT is a composite of three different algorithms: (i) the block sorting main engine, a lossless and very slightly expansive preprocessor; (ii) the move-to-front coder (MTF), a byte-for-byte simple, fast, locally adaptive noncompressive coder; and (iii) a simple statistical compressor, like a first order Huffman or arithmetic coding, doing the compression. Steps (ii) and (iii) work like a local optimization over the permuted text obtaining results similar to k-th order compression.

In [16] the block-sorting algorithm of the BWT is extended to word-based models, including other transformations, like *spaceless words* mentioned above, in order to improve the compression. Experimental results shows that the combination of word-based modeling, BWT and MFT-like transformations allows to obtain good compression effectiveness to be attained within reasonable resource costs.

4 Dictionary Mapping

In this section we propose a word-based scheme on PPM. Our objective has been carried out plugging an additional layer to precede PPM that replaces words by two byte codewords, and then these codewords will be codified with a conventional PPM.

According to Skibinski et al. [19] replacing words by codewords has advantages and drawbacks. First, the concept of replacing words with shorter codewords from a given static dictionary has at least two shortcomings:

1. The dictionary must be quite large (at least tens of thousands of words) and it is appropriate for natural language only.
2. No "high level" correlations, e.g. related to grammar, are implicitly taken into account.

In spite of these drawbacks, such an approach to text compression turns out to be an attractive one, and it has not been given as much attention as it deserves. On the other hand, the benefits of dictionary-based text compression schemes

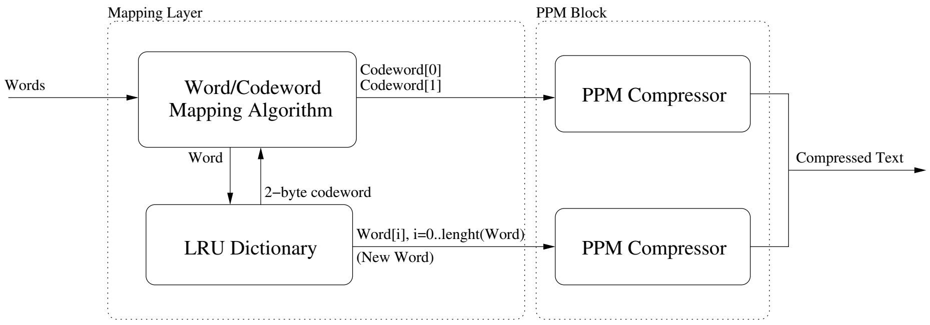

Fig. 1. Dictionary mapping

are: the ease of dictionary generation (assuming enough training text in a given language), clarity of ideas, high processing speed and acceptable compression ratios. Our proposal tries to solve both shortcomings.

Figure 1 shows a graphical representation of the dictionary mapping scheme. The proposal is made up of two different blocks: (1) Mapping Layer, which manages the vocabulary and maps words into codewords. To limit the number of different codes, a dictionary with a capacity of 2^{16} is used, when this dictionary is full an LRU policy is applied. (2) PPM Block, with two PPM compressors, which can be the same or not, one in charge of coding codewords and the other in charge of coding new words when they appear for the first time in the text.

This compression scheme can be seen as a PPC[1] compression scheme. This is a relatively new concept to compress data streams, based on the idea of pre-processing the data stream (through permutations and/or partitions) before compressing it. A successful PPC example is *bzip2*.

Algorithm 1 (Compression with dictionary mapping on PPM)

```
map.add(ESCAPE_WORD)
while (there are more words) do
        word ← get_word()
        if map.find(word) = true
           then
                 CodePPM_ENCODE(map.codeword(word))
           else
                 CodePPM_ENCODE(map.codeword(ESCAPE_WORD))
                 for 0 ≤ i < word.length() do
                     WordPPM_ENCODE(word[i])
                 od
                 map.add(word)
        fi
od
```

[1] Permutation–Partition–Compression

Algorithm 1 shows a generic scheme for compressing the text using dictionary mapping on PPM. This scheme uses two independent PPM encoders, one codifies the codewords and the other codifies the new words character by character. When a new word is reached, a reserved codeword (the general escape mechanism) is emitted, the new word is codified using another PPM encoder and it is added to the dictionary.

We handle a limited length size dictionary in order to always obtain two byte codewords, therefore the dictionary capacity is at most 2^{16} words. When the dictionary is full and it is necessary to insert a new word, the least-recently used (LRU) word is removed from the dictionary and its place is occupied by the new word. The idea behind this decision is to remove from the dictionary the words that have been used just once and probably they will never be used again. Since we are codifing natural language texts which obey the Zipf law [20] it is quite unlikely that a word will constantly be coming in and out the dictionary. Using this technique, we can represent any word of the vocabulary with two bytes and, consequently, an order-k PPM modelling codewords can better predict word sequences using the same amount of memory as another order-k PPM modelling the untransformed text.

The decompression algorithm is similar.

5 Evaluation

Tests were carried out on the SuSE Linux 9.3 operating system, running on a computer with a Pentium IV processor at 1.5 GHz and 384 megabytes of RAM. We used a **g++** compiler with full optimization. For the experiments we selected all the text files from Canterbury and Large corpora of the Canterbury Corpus[2] [2]. We also selected different size collections of WSJ, ZIFF and AP from TREC-3[3] [11]. In this case we concatenated files so as to obtain approximately similar subcollection sizes from the three collections, so the size in MB is approximate.

In order to test the dictionary mapping itself, and in conjunction with the two word-based techniques described in Section 2, we implemented several prototypes for basic dictionary mapping on PPM (denoted by *mppm*), dictionary mapping with separate alphabets model (denoted by $mppm_{sa}$) and dictionary mapping with spaceless words model (denoted by $mppm_{sw}$). In all the versions we used the Shkarin/Cheney's *ppmdi* [18] to obtain comparable results with the compressors mentioned below.

First we compressed the text files from Canterbury and Large corpora of the Canterbury Corpus. Table 1 shows the compression (in bits per character) obtained with our prototypes. The two first columns show, respectively, the file denomination and its size in bytes, whereas the third column shows the best results reported on the Canterbury Corpus site[4]. Column "ppmz" shows the

[2] `http://corpus.canterbury.ac.nz/`

[3] `http://trec.nist.gov/`

[4] `http://corpus.canterbury.ac.nz/details/`

compression obtained by Bloom/Tarhio's *ppmz v.9.1* for Linux[5], one of the better PPM variations but with high resources demand. Column "ppmdi" shows the compression obtained by the character based Shkarin/Cheney's *ppmdi*[6] (extracted in turn from James Cheney's *xmlppm v.0.98.2*). This *ppmdi* version uses the same memory requirements as *ppmdi* used to codify codewords in the *mppm* prototypes. In order to codify new words in *mppm* prototypes, another *ppmdi* compressor is needed but, in this case, with minor memory requirements. Word-based BWT compression was excluded because we could not find the software, yet results reported in [16] indicate that the compression ratios achieved for Canterbury Corpus are slightly worse to those of $mppm_{sw}$. Although, in order to be able to compare, it is necessary to make more tests, mainly with files of great size.

Table 1. Compression (bpc) for each method and collection, for Canterbury and Large corpora of the Canterbury Corpus

File	Size(bytes)	Best	*ppmz*	*ppmdi*	*mppm*	$mppm_{sa}$	$mppm_{sw}$
LIST	3,721	2.40	**2.253**	2.281	2.736	2.764	2.668
MAN	4,227	2.98	2.865	**2.852**	3.418	3.397	3.283
CSRC	11,150	2.08	1.867	**1.849**	2.293	2.329	2.244
HTML	24,603	2.32	2.192	**2.134**	2.382	2.473	2.355
PLAY	125,179	2.49	2.335	**2.307**	2.411	2.488	2.371
TEXT	152,089	2.20	**2.081**	2.033	2.145	2.189	2.090
TECH	426,754	1.95	1.827	**1.794**	1.861	1.873	1.834
POEM	481,861	2.36	**2.216**	2.253	2.267	2.344	2.266
WORLD	2,473,400	1.40	**1.295**	1.436	1.391	1.426	1.346
BIBLE	4,047,392	1.53	1.473	1.516	1.464	1.547	**1.436**

We can observe that the *ppmdi* compressor is better than the *mppm* prototypes for small sizes but worse for greater files (over 1 Mb), this is due to the overload when vocabulary is coded. On the other hand, $mppm_{sw}$ is the best of the *mppm* family and it is also the best choice for medium and large files, even improving on *ppmz* by 2.5% (as it uses memory without limitation). Comparing $mppm_{sw}$ with the best results reported in the site, it improves the compression from all files greater than 100 Kb. Also, all the prototypes of the *mppm* family fulfill this affirmation. In this collection, $mppm_{sw}$ improves *ppmdi* compression by up to 7%, *mppm* compression by up to 3.5% and $mppm_{sa}$ compression by up to 8%. $mppm_{sa}$ was expected to be superior to basic *mppm* for all files since it takes advantage of the alternation property. But it does not happen in most files in Table 1. This surprising behavior can be due to the fact that the ppm used in the basic *mppm* is able to predict longer sequences (including both words and separators) and therefore, it uses less bits in their codification than the $mppm_{sa}$,

[5] `http://www.cs.hut.fi/~tarhio/ppmz/`

[6] `http://pizzachili.dcc.uchile.cl/initiative.html`

which is composes of two PPM encoders, one for modelling words and the other for separators.

Next, we compressed different size collections of WSJ, ZIFF and AP from TREC-3 in order to verify the behavior of the algorithms when managing medium and large collections. TREC-3 collections are formed by semistructured documents, this can harm *mppm* compressors but allows us to compress documents with structure-aware compressors that obtain better compression than classical compressors. Therefore, we compressed the collections with several classic compressor systems: (1)*GNU gzip v.1.3.5*[7], which use LZ77 plus a variant of the Huffman algorithm (we also tried *zip* with almost identical results but slower processing); (2)*bzip2 v.1.0.2*[8], which uses the Burrows-Wheeler block sorting text compression algorithm, plus Huffman coding (where maximum compression is the default); (3)*ppmdi* (extracted from *xmlppm v.0.98.2*), the same PPM compressor used in *mppm* family and with the same parameters. This time, *ppmz* has been excluded due to its high memory and time requirements. However, to be able to have an idea of the *ppmz* behavior with TREC-3 collections, we have compressed the smallest collections obtaining a compression of 1.936 bpc for AP, 1.917 bpc for WSJ and 1.661 bpc for ZIFF. This compression ratio is just slightly better than the obtained by $mppm_{sw}$, but $mppm_{sw}$ demands much less resources. For longer texts, *ppmz* is simply not a choice.

On the other hand, we compressed the collections with other compression systems that exploit text structure: (1)*xmill v.0.8*[9] [14], an XML-specific compressor designed to exchange and store XML documents. Its compression approach is not intended to directly support querying or updating of the compressed documents. *xmill* is based on the *zlib* library, which combines Lempel-Ziv compression with a variant of Huffman. Its main idea is to split the file into three components: elements and attributes, text, and structure. Each component is compressed separately. (2)*xmlppm v.0.98.2*[10] [8], a PPM-like coder, where the context is given by the path from the root to the tree node that contains the current text. This is an adaptive compressor that does not permit random access to individual documents. The idea is an evolution over *xmill*, as different compressors are used for each component, and the XML hierarchy information is used to improve compression. (3)*scmppm v.0.93.3*[11] [1], that implements SCM, a generic model used to compress semistructured documents, which takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate model to compress the text that lies inside each different structure type. Like *xmlppm*, *scmppm* uses Shkarin/Cheney's *ppmdi* [18] compressors.

Table 2 shows the compression obtained with our prototypes for TREC-3 collections. We can observe that $mppm_{sw}$ is the best choice for the *mppm* family, improving $mppm_{sa}$ by up to 4% and mppm basic by up to 4.5%. Let us focus on

[7] `http://www.gnu.org`
[8] `http://www.bzip.org`
[9] `http://sourceforge.net/projects/xmill`
[10] `http://sourceforge.net/projects/xmlppm`
[11] `http://www.infor.uva.es/~jadiego`

Table 2. Compression (bpc) for each dictionary mapping prototype for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*mppm*	$mppm_{sa}$	$mppm_{sw}$	*mppm*	$mppm_{sa}$	$mppm_{sw}$	*mppm*	$mppm_{sa}$	$mppm_{sw}$
1	2.035	2.030	1.955	2.002	2.022	1.932	1.692	1.756	1.652
5	1.918	1.888	1.848	1.914	1.910	1.857	1.729	1.772	1.691
10	1.896	1.856	1.823	1.901	1.879	1.832	1.748	1.782	1.708
20	1.878	1.827	1.801	1.888	1.860	1.820	1.752	1.782	1.710
40	1.874	1.818	1.796	1.886	1.854	1.814	1.749	1.776	1.706
60	1.876	1.817	1.795	1.887	1.852	1.814	1.745	1.771	1.701
100	1.879	1.819	1.797	1.879	1.840	1.801	1.750	1.772	1.706

Table 3. Compression (bpc) for classical compressors for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*gzip*	*bzip2*	*ppmdi*	*gzip*	*bzip2*	*ppmdi*	*gzip*	*bzip2*	*ppmdi*
1	3.010	2.264	2.114	2.965	2.195	2.048	2.488	1.863	1.686
5	3.006	2.193	2.057	2.970	2.148	2.034	2.604	1.965	1.803
10	2.984	2.175	2.047	2.970	2.154	2.033	2.640	2.000	1.837
20	2.970	2.168	2.041	2.973	2.153	2.035	2.647	2.012	1.850
40	2.978	2.172	2.045	2.977	2.158	2.040	2.649	2.013	1.851
60	2.983	2.174	2.046	2.983	2.160	2.043	2.648	2.010	1.849
100	2.987	2.178	2.050	2.979	2.148	2.032	2.654	2.016	1.853

Table 4. Compression (bpc) for structure-aware methods for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*xmill*	*xmlppm*	*scmppm*	*xmill*	*xmlppm*	*scmppm*	*xmill*	*xmlppm*	*scmppm*
1	2.944	2.110	2.083	2.898	2.044	2.030	2.489	1.682	1.743
5	2.910	2.052	2.000	2.878	2.029	1.984	2.596	1.799	1.782
10	2.893	2.040	1.977	2.881	2.028	1.972	2.634	1.834	1.803
20	2.877	2.036	1.963	2.882	2.030	1.971	2.640	1.846	1.812
40	2.883	2.040	1.964	2.888	2.035	1.974	2.639	1.847	1.808
60	2.888	2.044	1.964	2.891	2.038	1.975	2.635	1.846	1.803
100	2.891	2.048	1.968	2.872	2.027	1.958	2.640	1.849	1.807

the $mppm_{sw}$ prototype in order to compare it with other systems. Compression for standard systems is shown in Table 3. The *gzip* obtained the worst compression ratios, not competitive in this experiment. It is followed by *bzip2* with the best compression as default and a great difference between it and *gzip*. The best standard compressor is *ppmdi*, the base for the *mppm* family, and with compression ratios near to *bzip2*. Our $mppm_{sw}$ prototype compressed significantly better than standard compressors. It improves *gzip* by up to 66%, *bzip* by up to 21% and *ppmdi* by up to 14%. Finally, in Table 4, we can see the compression obtained with structure-aware compressors for the same collections. *xmill* obtains an average compression roughly constant in all cases because it uses *zlib* as its main compression machinery, like *gzip*, its compression is not competitive in this experiment. On the other hand, *xmlppm* and *scmppm* obtain a good compression both surpassing standard compressors. However, in this case, our $mppm_{sw}$ prototype also still obtains the best compression, reaching an improvement on *xmill* of up to 54%, on *xmlppm* of up to 13.5% and on *scmppm* of up to 9.5%. In addition, $mppm_{sw}$ uses less memory than *xmlppm* and *scmppm*. In view of these results, we can conclude that $mppm_{sw}$ is an excellent alternative to compress natural language documents. A graphical representation of average compression is shown in Figure 2. In this graph we can observe that all the prototypes based on dictionary mapping are better (over 1 Mb in size) than all compressor systems against which they have been compared.

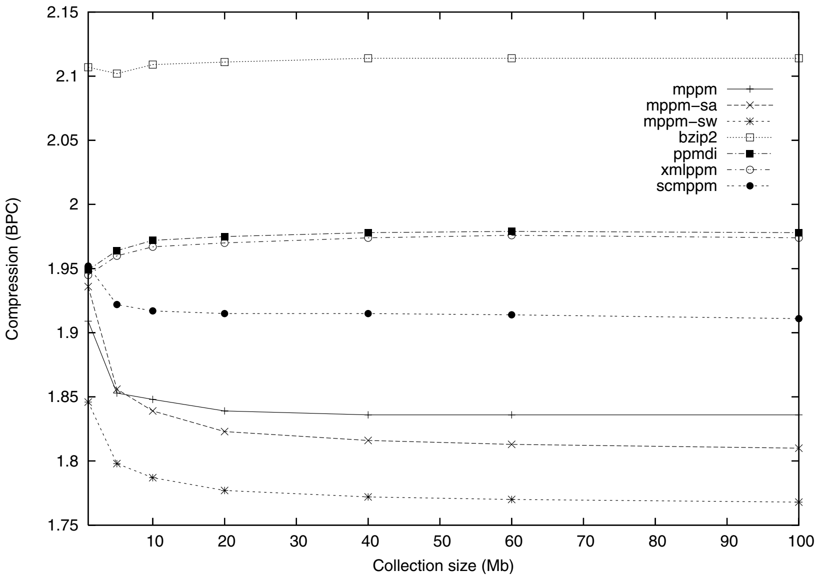

Fig. 2. Average compression for each TREC-3 collection size

The increase in space of the *mppm* prototypes with respect to *ppmdi* varies from 40% for $mppm_{sw}$ to 114% for $mppm_{sa}$. This increase is due to the necessity to store the vocabulary (dictionary) and to have an additional model to codify the new words. The $mppm_{sw}$ prototype is about 15% faster than *mppm* basic and both use approximately the same amount of memory. Besides for files up to 1 Mb, $mppm_{sw}$ is about 5000% faster than *ppmz* and uses 92% less memory than *ppmz*. That is why $mppm_{sw}$ is a very efficient alternative to *ppmz* for medium and large text files. On the other hand, $mppm_{sw}$ uses about 40% more memory and is also 40% slower than *ppmdi*. This increase in time is due to the time needed to locate a word in the used data structure (in this case a balanced search tree) this being $O(\log_2 n)$. It is possible to turn it into $O(1)$ if a hash table is used (where size can be estimated previously by using Heaps law[12]). The *ppmdi* obtains a constant average memory usage in all cases because it does not have to store the vocabulary.

6 Conclusions and Future Work

We have proposed a new, simple and efficient general scheme for compressing natural language text documents by extending the PPM to allow easy word handling using an additional layer. When file size grows, our proposal improves compression up to 14% with respect to the character based PPM. Our proposal uses just a little bit more memory and is a little slower, but these drawbacks are clearly compensated for by the gain in compression.

We have shown that the idea significantly improves compression and we have compared our prototype with standard and specific compressor systems, showing that our prototypes obtain the best compression for files over 1 Mb, improving the compression when file size grows. In addition, $mppm_{sw}$ is an interesting alternative for *ppmz* for natural language text files.

In this paper we have considered the compression of natural language text documents, and we will have to investigate the possibility of applying word mapping to binary files. We will have to generalize the mapping algorithm and we will have to avoid the generation of dynamic codes, as they prevent PPM from making good predictions. On the other hand, current mppm prototypes are a basic implementation and we are working on several improvements, which will make them even more competitive in terms of time and space.

References

1. J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Proceedings of 14th Data Compression Conference (DCC 2004)*, page 522, 2004.
2. R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of 7th Data Compression Conference (DCC 1997)*, pages 201–210, 1997.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley-Longman, may 1999.
4. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J., 1990.
5. T. C. Bell, A. Moffat, C. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44:508–531, 1993.
6. J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.
7. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

8. J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proceedings of 11th Data Compression Conference (DCC 2001)*, pages 163–172, 2001.
9. J. G. Clearly and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.
10. J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *Proc. ADBIS'99*, LNCS 1691, pages 75–84. Springer, 1999.
11. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.
12. H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, 1978.
13. R. N. Horspool and G. V. Cormack. Constructing word-based text compression algorithms. In *Proceedings of 2nd Data Compression Conference (DCC 1992)*, pages 62–71, 1992.
14. H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. ACM SIGMOD 2000*, pages 153–164, 2000.
15. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
16. A. Moffat and R. Yugo Kartono Isal. Word-based text compression using the Burrows–Wheeler transform. *Information Processing & Management*, 41(5):1175–1192, 2005.
17. E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proceedings of the Fourth South American Workshop on String Processing*, pages 95–111, 1997.
18. D. Shkarin. PPM: One step to practicality. In *Proceedings of 12th Data Compression Conference (DCC 2002)*, pages 202–211, 2002.
19. P. Skibinski, Sz. Grabowski, and S. Deorowicz. Revisiting dictionary-based compression. *Software–Practice and Experience*, 35(15):1455–1476, 2005.
20. G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison–Wesley, 1949.
21. N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.