Text Pre-processing for Lossless Compression

Luís Batista[1,2] and Luís A. Alexandre[1,2]
[1] Dept. Informática, Univ. Beira Interior, Portugal
[2] IT - Networks and Multimedia Group, Covilhã, Portugal

Textual data holds a number of properties that can be taken into account in order to improve compression. Pre-processing deals with these properties by applying a number of transformations that make the redundancy "more visible" to the compressor. One of the most commonly used concepts in text pre-processing is called Capital Conversion. Words with capital letters are converted to their lowercase versions while signaling the change with a flag. This way, not only context similarities are increased but also dictionaries used for word replacement only need to contain words in their lowercase versions. Word replacement consists of replacing words with shorter codes which are references to their location in a dictionary.
We propose the creation of "online" dictionaries presenting different alternatives for word orderings - by frequency, prefix, sufix and frequency x size-of-word - and a new implementation of the Capital Conversion technique. In our implementation of Capital Conversion, only two flags are needed to deal with words that have capital letters. Flags that encode words with first capital letter and totally uppercase words are used in the same way. When the word starts with a few uppercase letters and ends with lowercase letters, the uppercase flag is used for the first part and a flag that encodes a word with first capital letter is inserted before the last capital letter. We found that this is of great benefit since words that usually appear as a concatenation of two or more different words are now, most of the times, correctly separated. We also found that the contextual prediction also benefits with the use of only two different flags. Additionally, we reduce the use of escape flags needed to make the distinction between real flags and symbols that appear in files that aren't 100% text. Our method improved over existing methods.
Most schemes making use of word replacement must have the dictionary given in advance, making the approach language dependent and having the shortcoming of not always being suited for text files with specific vocabulary (ex. mathematics, programming, etc.). To avoid this, we propose the idea of creating the dictionary immediately before pre-processing and append it to the pre-processed file to be compressed. The cost of compressing the file plus the dictionary is compensated by the gain achieved by means of a tuned dictionary and the fact that the dictionary itself is compressed. The tests showed that dictionary word ordering has an impact in compression. From the different orderings we presented, the prefix dictionary proved to be the best for a 100Mbyte test file. It improved the compression by 2.8%. For small files, we consider only words that occur at least 25 times over the text to form the dictionaries.

Text Pre-processing for Lossless Compression

Luís Batista[1] and Luís A. Alexandre[1,2]
[1] Dept. Informática, Univ. Beira Interior, Portugal
[2] IT - Networks and Multimedia Group,
Covilhã, Portugal

Abstract

In this paper we present several pre-processing techniques developed to help general-purpose compressors achieve better results in the task of (lossless) text compression. The possibility to create dictionaries "online", together with the ability to store them within the compressed file, has revealed itself an attractive one, resulting in significant compression improvement. Moreover, this technique has the advantage of being independent for languages whose vocabulary is built upon the use of prefixes and sufixes. In our experiments, we achieved an improvement representing almost 3% over existing techniques on a large (100Mbyte) file.

1 Introduction

In recent years we have seen most state-of-the-art compressors become aware of some specific data-type properties that can be taken into account in order to improve compression. These compressors incorporate specialized algorithms to handle such data-types. In this paper we make an approach on one of those types: text. The two main paths, when it comes to text compression, are either the creation of specialized text compressors or the use of text pre-processing techniques. While the first surely has its own advantages, the fact that pre-processing can be used together with the vast majority of compressors makes it more appealing.
The use of pre-processing techniques can already boost compression gains up a few percent. A great number of text properties are handled by many

authors significantly in the same way, with minor variations. The conversion of words with capital letters to lowercase is one such example. Grabowsky [1] suggests the use of an escape symbol (flag) to encode the conversion. Abel and Teahan [2] consider for this conversion only words that occur elsewhere with lowercase letter and use a different flag for uppercase words. Skibinski *et. al.* [3] use yet another flag for words that have only a few capital letters. Still, minor modifications can sometimes have a great impact in compression, either directly, or by optimizing other algorithms. Capital conversion makes possible the use of only lowercase words in dictionaries meant for word replacement. As a result, they became smaller, a consequence of not having to store the same words in the different forms that may occur with capital letters.

Many schemes presented in literature making use of the idea of word replacement must have the dictionary given in advance [1, 3]. Besides the obvious limitation that such dictionaries make the approach language dependent, they also have the shortcoming of not always being suited for text files with specific vocabulary (ex. mathematics, programming, etc.). Nevertheless, on average, this approach produces very significant results and is of common use. Abel and Teahan [2], on the other hand, present a word replacement scheme based in dictionaries that is language independent. Words are added to the dictionary adaptively as they appear for the first time and are replaced with tokens in their subsequent appearances. They also consider a similar scheme for phrase replacement and a heuristic alphabet reordering scheme to group symbols with similar context. Grabowsky [1] uses a promising technique invented by Taylor, trying to reduce the effect caused by the end-of-line (EOL) symbols, which hamper the context, since words are usually separated by spaces. He substitutes EOL symbols by spaces and encodes their former positions.

In this paper we present improvements to existing ideas and also some new ideas. A change of the flag position in words that have a few capital letters is presented suppressing the need of an additional flag. We also consider different algorithms to produce dictionaries in a pre-pass over the text, embeddeding them in the file to be compressed. All this resulted in a gain reaching almost 2% over existing techniques.

The paper is organized as follows: the next Section contains a brief explanation of the common techniques in text pre-processing, Sect. 3 proposes our new approach, Sect. 4 presents the experiments and the final Section contains the conclusions.

2 Text pre-processing in lossless compression

The idea behind the text pre-processing approach is to apply a number of transformations to the original text file, converting it into another file (temporary file), which is more redundant. In fact, what it does is make the redundancy "more visible" to the compressor. Then, this new file can be compressed in the exact same way as we would compress the original one, but (hopefully) with better results. This process must be reversible, which means that, it must be possible to reverse the transformations so that, in the end, the decompressed file is an exact match of the original one.
Textual data holds a number of properties that are not always immediately visible to the compressor. Pre-processing deals with these properties using a series of algorithms that make the compressor become "aware" of them. Although there are a number of valid algorithms for text pre-processing, we will only consider for a brief review those used in our experiments. Whenever possible we establish a parallel with TextFilter (a pre-processor based in Skibinski's WRT [4]) which was the starting point for our work.

2.1 Capital Conversion

One of the most commonly used concepts in text pre-processing is called Capital Conversion. It is based in the fact that, for example, "Letter" and "letter" are not more than different representations of the same word, but are treated as distinct words by most compressors which, obviously, means worse compression ratios. In order to change this, words with capital letters are converted to their lowercase versions while signaling the change with a flag (usually a 1 byte flag). Among the advantages of using the capital conversion technique are the significant increase of context similarities and, when using word replacement, dictionaries only need to contain words in their lowercase versions.

2.2 Word Replacement

This well known technique is undoubtedly the pre-processing scheme that contributes the most to improve compression. It consists of replacing words in the text with shorter codes (code-words), whenever they are present in an external dictionary, i. e., words are replaced with references to where the word is located in the dictionary. The immediate advantage is that code-

words are shorter than words and, thus, the same amount of text will require less space. The size (in words) of the dictionary is limited to the number of code-words available. In TextFilter code-words can be formed by one up to four symbols (bytes) of the ASCII table above 128, since they are rarely used in text files. Not only it helps compressors in their predictions but is also of benefit when substituting word prefixes as we will see bellow. Still, not all of the 128 symbols can be used for every position in the code-word. Each position has its own sub-alphabet of symbols from which it takes values. In TextFilter, since code-words can have up to four bytes, four sub-alphabets have been formed: the first group has 64 symbols, the second 32, the third 16 and the fourth 16 symbols. Thus, words in the text matching one of the first 64 words in the dictionary will be replaced with code-words of length 1 byte, 2 bytes to the next $64 * 32$ words and so on. The most frequent words for a given language will be usually stored at the begining of the dictionary so that they will be given shorter code-words. Since the dictionaries are usually static, word frequencies are determined training in a large corpus from that language.

2.3 Prefix Replacement

Sometimes, it happens that a given word is not found in the dictionary but a smaller part of it is (e.g. the prefix of the word). A partial substitution can still occur. Since the alphabet for code-words is disjoint from the alphabet for original words, that part of the word can be replaced by the respective code-word without the need for any flag to mark the separation.

3 Our proposals

In this Section we propose some new ideas for text pre-processing as well as improvements to existing ideas. A text pre-processor was built from scratch to implement them. Although we based part of our work on what was done in Textfilter, we have decided to ignore many of its algorithms. The reason is that we focused on the creation and use of "online" dictionaries, and therefore were only interested in techniques that contribute in some way to the process of word replacement. We now present the implemented techniques.

3.1 Modified Capital Conversion

As we've seen above, capital conversion helps the process of word replacement so that only lowercase words need to be stored in the dictionaries. In our implementation only two flags are needed to deal with words that have capital letters as opposing to three flags in Textfilter. The two flags that encode respectively words with first capital letter and totally uppercase words are used in the same way. When the word starts with a few uppercase letters and ends with lowercase letters, the uppercase flag is used for the first part but, instead of using a third flag to separate it from the lowercase part, the same flag that encodes a word with first capital letter is inserted before the last capital letter. We found that this is of great benefit since words that usually appear as a concatenation of two or more different words are now, most of the times, correctly separated. We also found that the contextual prediction also benefits with the use of only two different flags. Additionally, we reduce the use of escape flags needed to make the distinction between real flags and symbols that appear in files that aren't 100% text, since we use one less flag.

3.2 Modified Word Replacement

The process of word replacement was also subject to changes. We decided to change the maximum size of code-words to three bytes instead of the previous four used in Textfilter. Consequently, the size of the byte groups used to build code-words has changed too. The choice of the new sizes was motivated by the work of Alexander Ratushniak in his (successful) attempts at the Hutter Prize [5]. He uses three groups with the following amount of bytes: 80 for the first group, 32 for the second group and 16 for the third, allowing that the second symbol of the code-word might be both from the second and third groups. This gives a total of $80 + 80 * 32 + 80 * 16 + 80 * 32 * 16 = 44880$ code-words available which, we believe, is an acceptable amount of words since the size of the dictionary has to be a compromise between replacing a fair amount of words in the text and creating a very large dictionary, since it will be stored with the compressed data. The choice of the group sizes is such that it is easy to distinguish to which group a certain symbol of a code word belongs to using only its most significant bits. Once a word of the text file is found in the dictionary, it is replaced by the respective code-word in the temporary file.

3.3 Online Dictionaries

Code-words are references to words in a dictionary so, in order to make Word Replacement, both the compressor and decompressor must have access to the same dictionary. If a static dictionary is used, which is the usual approach, it is assumed it will be delivered with the decompressor. To compress a different language text file, a different dictionary would be needed. The same would happen with other text files that use specific terms and vocabulary. In short, we would need as many dictionaries as all the languages and file types we wanted to compress. To avoid this, we propose the idea of creating the dictionary immediately before pre-processing and append it to the pre-processed file to be compressed. That way, a single relatively small decompressor does the job, without the need of extra dictionaries, many of which are rarely used. The cost of compressing the file plus the dictionary is compensated by the gain achieved by means of a tuned dictionary and the fact that the dictionary itself is compressed. Once we decompress the file, the dictionary can be detached and used to reverse pre-processing. For obvious reasons, we want the dictionaries to be built automatically. Several authors pointed out that the order in which words appear in the dictionary has an impact in compression. The two main reasons for it are: 1 - relatively big words become smaller (at least their representation) since they are replaced by shorter code-words; 2 - the number of context similarities can be greatly increased if words that usually appear in the same context are grouped together in the dictionary. Thus, the second question is how to order the words. The answer to this question is not as obvious as it may seem, and the solution we found was to produce differently ordered dictionaries. The following word ordering schemes were implemented.

3.3.1 Words ordered by Frequency

According to the scheme adopted for code-word attribution, the first 80 words in the dictionary will be replaced by 1 byte length code-words, the next $80 * 32 + 80 * 16$ words will be given 2 bytes and the remaining words 3 bytes tokens. The idea of ordering words by their frequency is that the most frequent words will be replaced by shorter codes. The resulting pre-processed file will therefore be smaller to the benefit of compression. A first pass is done while counting the number of occurrences of each word. The most frequent word will be the first in the dictionary, the second most frequent word will be

the second and so on until one of two things happen: either we have reached the end of the text file, and thus have already the least frequent word in the dictionary, or we have reached the limit of words for which there are code-words available, i. e., 44880 words.

3.3.2 Words ordered by Prefix

The reason for this type of word ordering is that words that start with the same prefix have a tendency to appear in similar contexts. As an example, the words "interregional", "intercontinental", "intertropical" or "interglacial" all suggest a certain relation between or among things, a similarity that is likely to be found in similar contexts. So the idea is to group them together in the dictionary so that they will be referenced by code-words whose first symbols are similar. That way, contexts are effectively extended and compressor predictions greatly improved. In our scheme to build these dictionaries, we do not use a truly alphabetic prefix ordering. Only the first four letters of each word (or less, if the word is smaller) are taken into account to form groups of words and even those groups are not sorted alphabetically. Instead, the first word is still the most frequent and the following words are the ones with the same prefix (four bytes). From the remaining words, again the most frequent is chosen followed by those with the same prefix. As before, we repeat the process until there are no more words or the dictionary is full.

3.3.3 Words ordered by Suffix

The procedure used for prefix ordering is also used for this type of dictionaries. The only difference is that instead of the prefix, it is the suffix (last four bytes) that is used. The justification is also the same: words with the same suffixes also tend to appear in similar contexts throughout the text.

3.3.4 Words ordered by Frequency x Size-of-Word

This scheme favors the substitution of less frequent words if their length compensates their count. Conversely, shorter words are favored if they are very frequent. The idea is similar to the Frequency ordering seen above with the advantage of providing a mechanism to "ensure" it is worth to trade the word by the code-word. The result is that some of the more frequent words will give their place to longer words while others will have their position swaped in the dictionary.

4 Experiments

To carry out our experiments we decided to use the PAQ8h [6] compressor. The PAQ8 series are, at the moment, among the best existing compressors and are ranking at the "top" of several compression competitions and benchmarks. This specific version of PAQ was chosen mainly because it already comes with a built in text pre-processor (Textfilter 3.0), against which, the results of our own pre-processor can be compared with. Part of our tests were made in the extract of the Wikipedia ENWIK8 [7], that is being used in the Hutter Prize [5] competition. The file size is 100Mbytes. We also made tests on some of the textual files of the Canterbury Corpus. The results are shown in Tables 1 and 2.

Our modification to the Capital Conversion scheme outperforms the existing TextFilter Capital Conversion scheme by 0.9%, as we can see on the ENWIK8 tests (Table 1). For the smaller files (Table 2), the results are equal for both schemes since in these files there is no situation where to apply the differences between these schemes.

We also see in Table 1 that the new group sizes for codewords (ENWIK8A) outperforms the old one (except for the sufix dictionary).

The tests showed that dictionary word ordering does in fact matter. From the different orderings we presented, the prefix dictionary proved to be the best for the large file. It improved the compression by 2.8% when compared to the original Capital Conversion scheme.

Our technique for creating dictionaries online didn't seem to perform very well on the small files. In fact, our first experiments showed that for these files we achieved better results if we didn't use it. The reason for this is that there are few word repetitions. Storing a word on the dictionary that only occurs once in the text results in adding size to the temporary file, which is the oposite of our objective. To overcome this problem, we introduced a small change in the creation of dictionaries meant for small files: to form the dictionaries, we consider only words that occur at least 25 times over the text. This way we improved results and surpass the ones obtained without the use of dictionaries for the ASYOULIK and PLRABN12. The case of LCET10 is interesting because none of these methods improves compression over the compression without pre-processing. We think this may be due to the fact that the file contains technical writing, which means it has a limited vocabulary where the use of dictionaries isn't compensated by the cost of having to store it within the file.

	ENWIK8	ENWIK8A
No Pre-processing	18.166.126	-
Capital Conv.	18.214.883	-
Mod. Cap. Conv.	18.198.049	-
Frequency	17.762.265	17.755.292
Prefix	**17.710.781**	**17.706.851**
Suffix	17.780.548	17.785.612
Count * Size	17.764.947	17.756.491

Table 1: Large file tests. Sizes after compression with PAQ8h. The column ENWIK8A represents the results obtained with the new group sizes for codewords. Best results are in bold. Sizes are in bytes.

	ALICE29	ASYOULIK	LCET10	PLRABN12
No Pre-proc.	35.223	33.176	**84.086**	122.274
Capital Conv.	**34.982**	33.130	84.090	122.137
Mod. Cap. Conv.	35.085	33.236	84.215	122.336
Frequency	36.284	34.165	88.100	126.127
Prefix	36.192	34.126	87.137	125.249
Suffix	36.261	34.142	87.905	126.129
Word count > 25				
Frequency	35.065	**32.920**	84.871	121.483
Prefix	35.041	32.928	84.830	121.481
Suffix	35.063	32.921	84.859	**121.476**

Table 2: Small file tests. Sizes after compression with PAQ8h. Best results are in bold. Sizes are in bytes.

5 Conclusion

In this paper we presented several techniques to improve lossless text compression. The proposals are pre-processing techniques. We showed that compression can be improved using dictionaries built "on-the-fly" and storing them within the compressed file. We discussed different alternatives of ordering the words in the dictionary and presented some improvements over existing pre-processing schemes. We are currently studying other pre-processing techniques as well as ways of predict how many times a word must occur in the text to make it worth to store it in the dictionary so that smaller files

can also benefit consistently from this scheme.

References

[1] Grabowski, S.: Text Preprocessing for Burrows-Wheeler Block Sorting Compression Proc VII Konferencja *Sieci i Systemy Informatyczne – Teoria, Projekty, Wdrozenia*, Lódź, Poland, 1999.

[2] Abel, J., Teahan, W.: Universal Text Preprocessing for Data Compression IEEE Trans. Computers, 54(5):497-507, 2005.

[3] Skibiński, P., Grabowski, S., Deorowicz, D.: Revisiting dictionary-based compression Software – Practice & Experience, 35(15):1455-1476, 2005.

[4] Skibiński, P.: Homepage of Przemyslaw Skibiński http://www.ii.uni.wroc.pl/~inikep/research/WRT/WRT46.zip, 2006.

[5] Hutter, M.: 50.000 Euro Prize for Compressing Human Knowledge http://prize.hutter1.net/, 2006.

[6] Mahoney, M.: The PAQ Data Compression Programs http://cs.fit.edu/~mmahoney/compression/paq8h.zip, 2006.

[7] Mahoney, M.: About the Test Data http://cs.fit.edu/~mmahoney/compression/enwik8.zip, 2006.