Text Preprocessing for Burrows-Wheeler Block Sorting Compression

Conference Paper · October 1999

CITATIONS
14

READS
239

1 author:

Szymon Grabowski
Lodz University of Technology
132 PUBLICATIONS **1,505** CITATIONS

VII Konferencja „Sieci i Systemy Informatyczne – teoria, projekty, wdroïenia”
Jydµ, paµdziernik 1999

SZYMON GRABOWSKI
Katedra Informatyki Stosowanej Politechniki Jydzkiej

TEXT PREPROCESSING FOR BURROWS-WHEELER BLOCK SORTING COMPRESSION

Abstract

Burrows-Wheeler block sorting algorithm is one of the most attractive compression methods. In this paper we present several ideas attempting to improve text compression while retaining the speed. All of them are performed before actual coding in an independent preprocessing phase which yields flexibility of the coder selection. Experiments with bzip, a well known block sorting implementation, indicate 2-4 per cent compression gains for typical English texts.

1. Introduction

The Burrows-Wheeler transform (BWT) compression algorithm is a relatively new method, especially appropriate for text and other data with stable statistics. Although it has been only five years since the method was presented, several outstanding implementations have already been released. In this paper we refer to the canonical post-BWT coding described in [1], however it should be mentioned this is not the only possible way.

The BWT transform takes all rotations of a given block of n bytes and sorts them lexicographically. The sorting groups similar contexts together and the last byte of each string is the transform output. The reversibility of the transform is shown eg. in [1, 3]. It is noteworthy that in most implementations the contexts consist of the bytes following the predicted character instead of more traditional preceding bytes. To compress the resulting data, the output of the transform is run through a Move-To-Front encoder and finally submitted to the arithmetic or Huffman coder.

Move-To-Front (MTF) coding outputs the rank of each byte in the sequence of its predecessors. Careful study of modelling and coding the output of BWT transform was presented in [3, 4].

Although conceptually simple, BWT-based compression algorithm (also called a block sorting coder or BS, for short) is not easy to be improved in general terms, that is,

without much assumption for input data. Its advantage over more traditional Prediction by Partial Matching (PPM) coders lies in seamless utilization of many similar contexts of different orders, the price is however lack of knowledge of actual contexts. What hurts most in block sorting coding is rapid changes in successive contexts resulting in high and costly MTF ranks. To relief rapid context changes, contexts with similar probability distributions should be placed close together. Ways to find appropriate alphabet orders for given data were proposed in [2] as well as a hard coded heuristic order chosen for English text was there presented. We use the heuristic order, in a slightly modified form, as one of the enhancements.

In this paper we also present another technique for removing certain rapid context changes introduced by line breaks and punctation marks used in typical text.

Converting capitals to lower case is also tested. It results in grouping intrinsically similar but formally different (and thus distant after BWT) contexts.

Yet another idea which we examine here is replacing frequent phrases in the given text with single characters outside the range of actually used characters.

Finally, we discuss a little the chances of successful coding of End-Of-Line characters.

2. Capital conversion

Capital letters preceding actual contexts usually result in high MTF ranks while their lower case equivalents would very often imply 0's in MTF ranks. For example, the (following) context „*he day...*" predicts *t*'s as well as *T*'s. We found it beneficial to replace such capitals with an equivalent lower case letter preceded by a unique flag. To improve compression, the conversion is omitted if the next character is not a lower case letter [6].

Example:

The Title → *#the #title*

but *THE TITLE* → *THE TITLE* (# – a flag)

Why does it work? Instead of quite a high MTF rank, we usually obtain two lower ranks, with one of them often being zero.

Below is an example. Table 1 refers to the original text. The current context is shown in the last row. '_' denotes a blank space.

Table 1

The current character's predecessor	current (predicted) character	following context
_	T	hen h...
_	W	hen i...
_	w	hen i...
_	w	hen i...
_	t	hen i...
_	t	hen s...
_	t	hen s...
_	T	hen t...

The MTF rank for 'T' in the last row is 3.

After the conversion instead of „*Then t...*”, we obtain „*#then t...*”, '#' being a flag (see Table 2).

Table 2

current character	context	current character	context
t	hen h...	#	then h...
t	hen i...	_	then i...
t	hen s...	_	then s...
t	hen s...	_	then s...
t	hen t...	#	then t...

The resulting MTF ranks are 0 (for 't' on the left) and 1 (for '#' on the right). According to Fenwick [3], MTF codes of 0, 1 and 3 for English texts cost generally about 0.5, 3 and almost 5 bit/symbol, respectively. It is clear now we can expect gain in the presented case.

We found that following the flag with a blank space in the described idea [9] is even slightly better on the overall, the improvement is however inconsistent, for example this is better on book1 and book2 but worse on bib (all files from Calgary Corpus). We decided to use the extended idea for the tests in Section 7.

To the best of our knowledge, the described capital conversion idea has never been presented before.

3. Space stuffing

Most text files have End-Of-Line characters (EOLs) followed by a letter. An EOL character (or the latter of the two EOL characters in CarriageReturn/LineFeed text style) is not predicted very well. To alleviate it, in the preprocessing phase we expand the data by adding a blank space at the beginning of each line. An added blank is predicted very well with the context starting with a full word and the character distributions for contexts starting with a letter are no longer adversely affected with the EOL character. In other words, it is better to sacrifice prediction of contexts starting with a blank space for improving prediction of contexts starting with almost any other character.

An additional little improvement is to refrain from adding spaces after EOLs if the next character is neither a letter nor a space [5]. It can help eg. on program sources with many lines starting with a tab character.

Some texts have almost almost all lines started with one or more spaces (eg. plays, poems). A careful implementation could detect such cases and then refrain from adding spaces after EOLs for the whole file/block.

4. Phrase substitution

Replacing frequent phrases in the original data with new symbols from an augmented alphabet before actual compression is a known idea [8], we however are not aware of any application of this concept to block sorting compression. Teahan reports also increased memory requirements for PPM after phrase substitution due to expanded alphabet size. This should not be a problem for BS, since most sorting algorithms for BWT transform require memory resources proportional to the block size, and not depending on its contents as long as a single symbol in the augmented alphabet occupies no more bits than in the original alphabet. Plain ASCII text uses 7 bits only, so the natural choice was not to exceed 256 symbols in the expanded alphabet. Possible collisions (for non-„pure ASCII" data) must be handled by data expansion. In a practical implementation for general usage, data type detection should be performed before using phrase replacement. For example, English text might be detected by finding a reasonable amount of the word „the" in the given file/block; it is also recommended to estimate the rate of non-ASCII characters in the file/block. If it does not exceed eg. 5 per cent, we could feel confident the phrase replacement will not hurt much. The set of phrases may be static (hard coded) or selected dynamically for given data. From a practical point of view, dynamic phrase substitution offers little for its increased complicacy and computational costs. We chose to test static phrase substitution only. Our phrase set was selected for English language, since it is estimated that 80% of all texts on the Internet are in English [8].

The advantage of phrase substitution in block sorting compression comes from removing certain runs of zeroes in MTF values as well as managing to comprise more actual data in one block. The negative impact is context decorrelation. For example, the string „*tha...*” might be placed far from the string „*the...*” if „*the*” is in the set of substituted phrases. This undesirable phenomenon may be mitigated by proper alphabet reordering (see Section 5).

It is not easy to give a rule to finding the most appropriate phrases to be replaced. One simple and rather obvious remark is that certain frequent phrases may not really be good candidates for the phrase set. For example, selecting the phrase „*ther*” (prefix of the word „*there*”) seems inappropriate while the phrase „*the*” is to be selected.

Our set was found experimentally. We limited the set to phrases of length two (digrams), three (trigrams) and four. The text was scanned three times: in the first pass the phrases of length 4 were substituted while in the last pass digrams were replaced. Using lookup tables enabled to perform the replacement very fast. All phrases consisted of lower case letters only and the 4-character phrases had all unique 3-character prefixes, so the memory requirements for the lookup tables were modest.

Below we present our set of phrases (Table 3). It consists of 9 quadgrams, 26 trigrams and 88 digrams.

Table 3

4-char phrases	trigrams	digrams
that	*all*, *and*, *but*,	*ac*, *ad*, *ai*, *al*, *am*, *an*, *ar*, *as*, *at*,
said	*dow*, *for*, *had*,	*ea*, *ec*, *ed*, *ee*, *el*, *en*, *er*, *es*, *et*,
with	*hav*, *her*, *him*,	*id*, *ie*, *ig*, *il*, *in*, *io*, *is*, *it*,
have	*his*, *man*, *mor*,	*of*, *ol*, *on*, *oo*, *or*, *os*, *ou*, *ow*,
this	*not*, *now*, *one*,	*ul*, *un*, *ur*, *us*, *ba*, *be*, *ca*, *ce*, *co*, *ch*,
from	*she*, *the*, *was*,	*de*, *di*, *ge*, *gh*, *ha*, *he*, *hi*, *ho*,
whic	*wer*, *whi*, *whe*,	*ra*, *re*, *ri*, *ro*, *rs*, *la*, *le*, *li*, *lo*, *ld*, *ll*, *ly*,
were	*wit*, *you*, *any*,	*se*, *si*, *so*, *sh*, *ss*, *st*, *ma*, *me*, *mi*,
tion	*are*	*ne*, *nc*, *nd*, *ng*, *nt*, *pa*, *pe*,
		ta, *te*, *ti*, *to*, *th*, *tr*, *wa*, *ve*

Two digrams from the selected set, „*th*” and „*on*”, were split ie. each of them was substituted with one of two symbols depending on if the character preceeding the phrase were a blank or not. Actually, the substitution introduced 9+26+88+2=125 new symbols.

5. Alphabet reordering

The degree of similarity of neighboring contexts depends on the sorting order. Any alphabet order is allowed as long as both the coder and the decoder know it. It turns out the the alphabetical ASCII order is not the best choice. More information on finding a better alphabet order, also dynamically for the given data, can be found in [2], to the best of our knowledge the first and so far only study of the aspect of alphabet reordering for BWT. In our tests we use a modified version of the heuristic order for English language taken from that paper. Adaptive strategies there presented yield less consistant (and not really greater on the overall) improvements and their usage is limited due to the long time of the search. The heuristic order from the mentioned paper groups vowels, similar consonants, and punctation. For the lower case letters, the order is: „aeioubcdgfhrlsmnpqjktwvxyz" and is analogous for the upper case. The other groupings are „?!" and „+-,.". It performs well on text, and, since the order is similar to the original ASCII order, it doesn't usually hurt much on non-text data.

Our modification to the heuristic order intermingles digrams chosen for the phrase substitution with original symbols (eg. „*th*" is put close to '*t*'). We additionally group „special characters" (punctation marks, EOL characters, space, capital flag etc.).

An intrinsically similar idea to grouping punctation, EOLs etc. together is adding spaces before each of several chosen punctation marks. It is clearly an extension of the idea of adding spaces after EOLs from Section 3. The key observation is that it is a widely respected convention not to precede a dot neither a comma with a blank space in typed text; instead they are followed with a blank. The convention is not equally strict for other punctation marks, nevertheless it seems reasonable to precede some of them with a blank as well.

Which of those two almost equivalent concepts is better? We chose symbol grouping and gave up adding spaces before punctation for containing more actual data in BS blocks and slightly faster preprocessing.

6. EOL coding

End-Of-Line characters are hard to predict. If we had spaces instead of EOLs, most text files would be noticably more compressible. Hence comes the idea of replacing EOLs with spaces and grouping the EOLs together represented as line lengths in words (more precisely: each EOL is represented as the number of blank spaces since the previous EOL) [7]. Then all data („real" data and EOL information) together are submitted to BWT. This idea has unstable influence: on some files (eg. book1 from Calgary Corpus) it gives noticable improvement, on other files it hurts. The following order-1 entropy measurement heuristic was tested. The sum of sizes of order-1 arithmetically compressed actual data with EOLs replaced with spaces and order-1

arithmetically compressed EOL information was estimated and compared to the size of original data arithmetically compressed in the order-1 model. If and only if the sum of compressed sizes was lower, then the described EOL coding and reordering was performed.

The heuristic gave limited success. We decided to stop exploring the EOL coding idea but it looks promising for further research. For example, the EOLs can be encoded separately with eg. order-1 arithmetic coder which seems to give better results (we however limited ourselves in this work to mere preproccessing only so that solution is out of our scope). Another interesting question is about proper representation of EOL information. In many text files line lengths (in characters) are similar (ie. fixed left and right margins are kept). This information could be explored.

We would like to mention that our result on book1 with all ideas including the EOL coding and submitting all data to one BWT block was about 220300 bytes with bzip. This is, as can be seen below, over 2000 bytes additional gain. Book1 was however the only file from Calgary Corpus for which the EOL coding was helpful.

7. Test results

The presented filters are applied in the following order:

- capital conversion (to increase the number of phrase substitutions in the next step);
- phrase replacement;
- alphabet reordering (the new symbols for phrases are intermingled with original symbols);
- space stuffing (with respect to the new alphabet order) if EOL coding is to be omitted,

otherwise

- EOL coding if the order-1 entropy check heuristic suggests it. If the decision is negative, space stuffing (ie. adding spaces after EOLs) may then be used.

We did not use the EOL coding in our test.

In Table 4 we present the results on 14 text files from the Calgary Corpus obtained with *bzip* and IMP. *Bzip*, by Julian Seward, is a well-known block sorting compressor, based on Fenwick’s research in this area; it is available with sources [10]. IMP, from Technelysium Pty Ltd, is a very fast and relatively new implementation of block sorting compression [11]. The main difference between them is that *bzip* uses arithmetic coder while IMP uses Huffman coder. Each of the tested files from any of the two following corpora was contained in one *bzip*’s or IMP’s block.

Table 4

File	bzip (original data)	bzip (after filtering)	gain (%)	IMP -2 (original data)	IMP -2 (after filtering)	gain (%)
bib	27097	26839	0.95	27549	27264	1.03
book1	230247	222567	3.34	232103	223962	3.51
book2	155944	150291	3.63	157155	150933	3.96
news	118112	114927	2.70	118598	115009	3.03
paper1	16360	15772	3.59	16645	16073	3.44
paper2	24826	23839	3.98	25203	24152	4.17
paper3	15704	15194	3.25	16005	15481	3.27
paper4	5072	4881	3.77	5291	5089	3.82
paper5	4704	4570	2.85	4934	4760	3.53
paper6	12132	11666	3.84	12480	11981	4.00
progc	12379	12086	2.37	12668	12377	2.30
progl	15387	15023	2.37	15774	15435	2.15
progp	10533	10376	1.49	10855	10708	1.35
trans	17488	17142	1.98	17801	17499	1.70
total	665985	645173	3.12	673061	650723	3.32
avg.			2.86			2.95

As it can be seen, compression was never hurt on tested files. The overall gain is about 3 per cent, a little greater for IMP. On plain texts the improvement for IMP was usually greater (eg. book1, book2, news), on more structural texts (eg. progc, progp, progl, trans) the opposite seemed to be true. In other words, IMP was more sensitive to text preprocessing.

With both compressors more gain was obtained for plain, human language files than for more structural, „artificial" files.

Since the presented filtering was tuned to the Calgary Corpus files, with slightly more attention to plain text (book1) than to other text types, it should be interesting to evaluate the performance of filtering on another set of files. We chose the Canterbury Corpus (Table 5). Again, we skip the files containing characters with ASCII codes > 127.

Table 5

File	bzip (original data)	bzip (after filtering)	gain (%)	IMP -2 (original data)	IMP -2 (after filtering)	gain (%)
alice29	42823	41159	3.89	43321	41601	3.97
fields	2911	2829	2.82	3114	3076	1.22
lcet10	106629	102394	3.97	107208	102711	4.19
plrabn12	144223	141491	1.89	145467	142102	2.31
cp	7517	7350	2.22	7750	7577	2.23
grammar	1184	1161	1.94	1348	1338	0.74
xargs.1	1655	1595	3.63	1819	1769	2.75
asyoulik	39315	38454	2.19	39823	38857	2.43
total	346257	336433	2.84	349850	339031	3.09
avg.			2.82			2.48

As before, most gain is achieved on plain texts (lcet10, alice29).

8. Conclusions

The appeal of BWT-based compression comes from its speed comparable to typical LZ coders combined with superior compression rates for text data, only several per cent worse than those achieved with the best existing text compressors today, RKIVE by Malcolm Taylor [12] and BOA by Ian Sutton [13]. We showed three ways to further improve text compression of blocksorters. „Context alleviation” (space stuffing and capital conversion) and phrase substitution are of more general usage. The former helps to PPM and the latter helps to both PPM and, above all, LZ schemes. Alphabet reordering, which we used for additional slight gain, is BWT specific technique, although we noticed a slight improvement also for the *associative coder* by George Buyanovsky (ACB), a very interesting algorithm having much in common with blocksorters [14].

The overall gain for *bzip* and for IMP, two notable block sorting implementations, was about 3 per cent, which is a significant step toward the best known compression algorithms. Our preprocessing was fast and the overall compression speed was almost unaffected. In special cases, the speed could even be increased since the phrase substitution outputs a noticably smaller volume of data for further coding. A too large set of substituted phrases may hurt compression while making it faster in the same time. It is clear that fact raises a speed / compression trade-off. We also would like to stress that the considered ideas are independent from the used block sorting coder, in the

sense we may perceive the actual coder as a black box. Hence the simplicity and ease of application comes.

The main difficulty with our ideas is the fact they are not general. Context alleviation helps with most human languages but not necessarily with other structured texts like computer programs in various languages. The alphabet order was optimized for English, although, for the positive side, some key observations supporting the chosen symbol permutation are common for many languages. Phrase replacement is also language specific. Moreover, the most frequent phrases for different texts in one language may vary significantly. On the other hand, shorter phrases, particularly digrams, are more or less equally common within one language. Our experiments showed that replacing phrases longer than 3 characters does not give much additional benefit, so we restricted ourselves to digrams, trigrams and a few 4-character phrases only. It also enabled an easy and fast implementation of this idea.

In practical application, for each of the ideas lacking robustness, it is recommended to perform a preliminary scan with a quick heuristic trying to determine if the idea in question is applicable. Several simple hints were given in appropriate parts of this paper but they were not implemented. Our purpose was to demonstrate that the presented simple ideas often give significant benefits for compression and are worth further exploration rather than dealing with heuristics for handling special cases. Therefore the question about the range of applicability of those ideas remains for further research.

Acknowledgements

The author would like to thank Malcolm Taylor, Uwe Herklotz, George Lyapko, Ian Sutton and Vadim Yoockin for valuable suggestions, ideas and interesting discussion.

References

[1] M. Burrows and D. J. Wheeler, *A block-sorting Lossless Data Compression Algorithm*, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994. Also available at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html`.

[2] B. Chapin and S. R. Tate, *Higher Compression from the Burrows-Wheeler Transform by Modified Sorting*, University of North Texas, Department of Computer Science, 1998. Also available at `http://www.cs.unt.edu/~srt/papers/bwtsort.pdf`.

[3] P. Fenwick, *Experiments with a Block Sorting Text Compression Algorithm*, The University of Auckland, Department of Computer Science, Tech. Rep. 111, May 1995. Also available at `ftp://ftp.cs.auckland.ac.nz/out/peter-f/TechRep111.ps`.
[4] P. Fenwick, *Block Sorting Text Compression - Final Report*, The University of Auckland, Department of Computer Science, Tech. Rep. 130, April 1996. Also available at `ftp://ftp.cs.auckland.ac.nz/out/peter-f/TechRep130.ps`.
[5] U. Herklotz, private correspondence, 1999.
[6] G. Lyapko, private correspondence, 1999.
[7] M. Taylor, private correspondence, 1998.
[8] B. Teahan, *Modelling English text*, PhD thesis, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, May 1998.
[9] V. Yoockin, private correspondence, 1999.
[10] Bzip, version 0.21, sources available at `ftp://ftp.elf.stuba.sk/pub/pc/pack/bzip021.tgz` or `ftp://ftp.elf.stuba.sk/pub/pc/pack/bzip021.zip`.
[11] IMP, version 1.1, available at `http://www.technelysium.com.au/`.
[12] RKIVE, version 1.92 beta 1, available at `http://www.geocities.com/SiliconValley/Peaks/9463/`.
[13] BOA, version 0.58 beta, available at `ftp://ftp.elf.stuba.sk/pub/pc/pack/boa058.zip`.
[14] ACB version 2.00c, available at `ftp://ftp.elf.stuba.sk/pub/pc/pack/acb_200c.zip`.