

Revisiting dictionary-based compression

Przemysław Skibiński*, Szymon Grabowski†, Sebastian Deorowicz‡

January 15, 2005

This is a preprint of an article accepted for publication in
Software—Practice and Experience
Copyright © 2005 John Wiley & Sons, Ltd.
<http://www.interscience.wiley.com>

Abstract

One of the attractive ways to increase text compression is to replace words with references to a text dictionary given in advance. Although there exist a few works in this area, they do not fully exploit the compression possibilities or consider alternative preprocessing variants for various compressors in the latter phase. In this paper, we discuss several aspects of dictionary-based compression, including compact dictionary representation, and present a PPM/BWCA oriented scheme, *word replacing transformation*, achieving compression ratios higher by 2–6% than state-of-the-art StarNT (2003) text preprocessor, working at a greater speed. We also present an alternative scheme designed for LZ77 compressors, with the advantage over StarNT reaching up to 14% in combination with `gzip`.

KEY WORDS: lossless data compression, preprocessing, text compression, dictionary compression

1 Introduction

The situation in the field of lossless data compression in recent years has become rather strange. Most scientific papers still deal (or rather are said to deal) with the so-called universal compression, which can approximately be characterized by two features: (*i*) no specific assumptions about the input data; (*ii*) adhering to the Markovian statistical model. In the real world, however, most existing compressors no longer satisfy these conditions. Firstly, practically any modern compressor from the prediction by partial

*University of Wrocław, Institute of Computer Science, Przesmyckiego 20, 51–151 Wrocław, Poland, E-mail: inikep@ii.uni.wroc.pl

†Technical University of Łódź, Computer Engineering Department, Al. Politechniki 11, 90–924 Łódź, Poland, E-mail: sgrabow@zly.kis.p.lodz.pl

‡Silesian University of Technology, Institute of Computer Science, Akademicka 16, 44–100 Gliwice, Poland, E-mail: Sebastian.Deorowicz@polsl.pl

matching (PPM) [1] or Burrows–Wheeler transform (BWT) [2] family does not fit the Markovian model (for example, such concepts as information inheritance [3] in PPM, inverted frequencies transform [4] after the BWT, or encoding based on weighted probabilities [5]). Even the mere BWT makes a compressor based on it practically unable to stick precisely to this model. Although a candidate to dethrone Markov is desperately needed, we are not aware of any serious proposals. Secondly, more and more data-dependent “tricks” are in common use. They explore specific properties of text [6, 7, 8, 9], XML [10, 11], platform-dependent executable files [12], record-aligned data [13, 14], and multimedia data, to name the most important. It is interesting that several existing compressors incorporate specialized algorithms for some kinds of data that surpass the “official” state-of-the-art algorithms for the respective data types. The gap between the effectiveness of the algorithms presented in literature, especially the so-called universal ones, and those implemented in the best programs, is constantly increasing. This phenomenon has been the main motivation for our work.

In this paper, we decided to examine the impact of specialized modeling ideas on compression of textual files. There are two distinct approaches to text compression. One is to design a “text aware” compressor, the other is to write a text preprocessor (filter) which transforms the original input into a representation more redundant for general-purpose compressors. Specialized text compressors are potentially more powerful, both from the viewpoint of compression performance and the compression speed at a given compression rate, as there are virtually no restrictions imposed on the implemented ideas. Nevertheless, text preprocessing is more flexible, as the filtered text can be well compressible with most existing (and hopefully future) general-purpose compressors, so with relatively little programming effort various compression speed/decompression speed/compression ratio compromises can be achieved.

Several specialized text compressors have been presented so far. We would like to mention WordHuff [15], word-based LZW [16, 17], word-based PPM [18, 19], and BWT compression on words [20]. Text filtering ideas have been described by, among others, Teahan and Cleary [21], Kruse and Mukherjee [22], Grabowski [7], Awan *et al.* [6], Smirnov [23], Sun *et al.* [8], and Abel and Teahan [9].

One of the simplest, yet powerful concepts of boosting compression on a limited class of sources (not necessarily textual) was introduced by Teahan and Cleary [21]. They run PPM compression with a prebuilt model suitable for given data; in other words, PPM in their scheme starts compression with lots of useful knowledge. Although this idea has been definitely promising (cf. also Reference [19], pp. 133–139), the main problem was to build a model performing well on a relatively broad set of text files, for example. Recently an outstanding implementation of this concept was presented by Shkarin. His compressor, *Durilca* [24], incorporated several built-in models, selected for given data via a data detection routine. What makes this scheme similar to the preprocessing approach is employing the PPM engine directly without any modification; an apparently attractive feature. What makes it different from preprocessing techniques, and is another benefit, is that Shkarin’s idea needs a single pass over input data, while the “preprocess-and-compress” approach requires at least two passes. *Durilca*, together with its faster version, *Durilca Light*, achieves excellent compression, but the main drawback of this approach is the limitation to PPM family.

Our paper focuses on text preprocessing. We try to make use of almost all known text preprocessing techniques to achieve better compression and also propose a few novel ideas. As a result, our advantage in compression over the competition reaches a few percent, or sometimes even more. We also take care about encoding and decoding speeds which makes the presented scheme attractive even in combination with fast compressors, e.g., from the Lempel–Ziv family.

The organization of this paper is as follows. In Section 2, we briefly review the related works, focusing on StarNT algorithm developed by Sun *et al.* [8], which was the starting point of our research. Section 3 contains the description of our algorithm, word replacing transformation, designed for modern compressors from PPM and BWCA (Burrows–Wheeler compression algorithms) families, stressing the differences and extensions to StarNT. The next section presents the comparative results of extensive experiments with several compressors following different approaches (LZ77, BWCA, and PPM). In Section 5, an alternative preprocessing variant for LZ77 compressors is presented, together with the relevant experimental results. Sections 6 and 7 reveal the implementation details of our scheme, including a discussion of convenient techniques for dictionary search: ternary search trees, chained hashing, and minimal perfect hashing with finite state automaton. The effect of reducing the dictionary, desirable in some practical settings, is discussed in Section 8, both in the aspect of the resulting text compression and in the size of the compacted dictionary. Finally, several conclusions are drawn and perspectives for the future are suggested.

2 Related work

It appears that the so-called universal compression is nowadays reaching a plateau. Hence, in the research community a raise of interest toward specialized compression algorithms, including text compressors, can be observed. In the introduction, we outlined the main research avenues and here would like to concentrate on text preprocessing ideas.

Teahan and Cleary [21] discussed and tested several variants (mostly not presented earlier in the literature) of enlarging the alphabet with frequent English q -grams. They noted that extending the alphabet only with a single unique symbol denoting the **th** digram, improves PPM compression by up to 1%. Chapin and Tate [25] proposed alphabet reordering for BWCA compressors, with the aim of alleviating rapid context changes. This idea, notwithstanding its originality, is specific to a single family of compressors, and the gains are usually below 1%. Grabowski [7] presented several simple ideas, old and new, some of which will be described in Section 3, and reached an almost 3% gain with **bzip** [26] on the Calgary corpus text files. Although the filters were designed for and tested with BWCA compressors, most of them, in one or another form, are applicable to other compressors as well. In a recent work, Abel and Teahan [9] examined carefully the interplay between the existing text preprocessing methods and three different compressors. They added several enhancements and reached from 3% to 5% average improvement on the text files from the Calgary corpus. Their approach is language-independent, which in particular means that using a hard-coded set of q -grams is forbidden, but the actual algorithm version they present is limited to the Latin alphabet. In the light of language

independence, the mentioned gains in Abel and Teahan’s work are significant, and their most successful idea was to replace the most frequent words and q -grams in the given text with extra symbols. It should be stressed that the interest in text preprocessing is even greater in existing compressors than in literature: to name only a few, RKC [27] by Taylor, PPMN [28] by Smirnov, DC [29] by Binder, and **Compressia** [30] by Gringeler, but the full list is much longer.

All the aforementioned ideas are “lightweight”, in the sense that they are easy to implement and do not require significant amounts of extra memory or laborious preparation of auxiliary data. On the downside, their resulting gain in compression ratio is limited to about 5%, and even less with more advanced compressors. In this category of preprocessing techniques, we can also include the unpublished idea of removing spaces between words, converting words’ starting letters to uppercase (while signaling the original capitalized words with a flag), and possibly performing a, necessarily restricted in this case, q -gram replacement. In combination with **gzip**-like compressors from the LZ77 family, the obtained gains may reach about 7–8%, but the improvements vary significantly from file to file.

Still, much more significant improvements are possible with more complex schemes, most notable being based on the notion of replacing whole words with shorter codes. The dictionary of words is usually static, for a given language, and available in advance, as building the dictionary for given input data—“on the fly” or in a prepass—is more difficult, often slower, and, most importantly, yields worse compression ratios.

Kadach [31] in his Ph.D. dissertation dates back the first word replacing compression scheme to 1963 and lists 17 relevant publications. His own algorithm in this domain is not only preprocessing as it uses specific coding of word indices. The dictionary of all words used in a given text is kept within the compressed output, and the overall compression results are quite impressive in this class, about 20% better than from a zip compressor at double the speed.

Teahan [19] considers two variations of word substitution methods. One encodes so-called function words (articles, pronouns, etc.), the other simply encodes about two hundred most frequent words. The frequent word encoding was more beneficial, still the gain to pure PPM was in the order of 2% only.

Kruse and Mukherjee [22] devised a dictionary-based scheme called Star encoding. They replace words with sequences of * symbols accompanied with references to an external dictionary. The dictionary is arranged according to word lengths, and the proper subdictionary is selected by the length of the sequence of “stars”. There have been several minor variations of such a scheme from the same authors, most popular of which is a length index preserving transformation (LIPT) [6]. In LIPT, the word-length-related subdictionary is pointed by a single byte value (as opposed to a sequence of “stars”). Still, the algorithm improves the PPM or BWCA compression by only about 4–5%, while with **gzip** the gain is less than 7%.

Smirnov [23] proposed two modifications to LIPT. One is to use non-intersecting alphabet ranges for word lengths, word indices, and letters in words absent from the dictionary. This idea gave a 3% gain with **bzip2**. The other idea is more complex: apart from non-intersecting alphabets, also more subdictionaries are considered, determined now not only by word lengths but also part-of-speech tags. The latter variant resulted

in about a 2.5% gain over the original LIPT with a PPM compressor, while with `bzip2` it was less efficient. For an LZ77 compressor, the original LIPT performed best.

StarNT [8] is the most recent algorithm from the presented family. In terms of compression, it brings 2–5% improvement compared with LIPT when a PPM or BWCA compressor is involved, and even about 10% with `gzip`. A word in StarNT dictionary is a sequence of symbols over the alphabet `[a..z]`. There is no need to use uppercase letters in the dictionary, as there are two one-byte flags (reserved symbols), f_{cl} and f_{uw} , in the output alphabet to indicate that either a given word starts with a capital letter while the following letters are all lowercase, or a given word consists of capitals only. Another introduced flag, f_{or} , prepends an unknown word. Finally, there is yet a collision-handling flag, f_{esc} , used for encoding occurrences of flags f_{cl} , f_{uw} , f_{or} , and f_{esc} in the text.

The ordering of words in the dictionary D , as well as mapping the words to unique codewords, are important for the compression effectiveness. StarNT uses the following rules:

- The most popular words are stored at the beginning of the dictionary. This group has 312 words.
- The remaining words are stored in D according to their increasing lengths. Words of same length are sorted according to their frequency of occurrence in some training corpus.
- Only letters `[a..zA..Z]` are used to represent the codeword (with the intention to achieve better compression performance with the backend compressor).

Each word in D has assigned a corresponding codeword. Codewords' length varies from one to three bytes. As only the range `[a..zA..Z]` for codeword bytes is used, there can be up to $52 + 52 \times 52 + 52 \times 52 \times 52 = 143,364$ entries in the dictionary. The first 52 words have codewords: `a, b, ..., z, A, B, ..., Z`. Words from the 53rd to the 2756th have codewords of length 2: `aa, ab, ..., ZY, ZZ`; and so on.

3 Word Replacing Transformation

The concept of replacing words with shorter codewords from a given static dictionary has at least two shortcomings. First, the dictionary must be quite large—at least tens of thousands words—and is appropriate for a single language only (our experiments described in this paper concern English text only). Second, no “higher level”, e.g., related to grammar, correlations are implicitly taken into account. In spite of those drawbacks, such an approach to text compression turns out to be an attractive one, and has not been given as much attention as it deserves. The benefits of dictionary-based text compression schemes are the ease of producing the dictionary (assuming enough training text in a given language), clarity of ideas, high processing speed, cooperation with a wide range of existing compressors, and—last but not least—competitive compression ratios.

In this section, we present our modifications to StarNT, as well as a few new ideas. The results of our modifications, step by step, can be observed in Table 1. From now on,

Table 1: Improving StarNT step by step on `book1`. The compression ratios are expressed in bits per input character. (The used compressors’ switches are described in Section 4.)

Method	PAQ6	PPMonstr	UHBC	bzip2	gzip
No preprocessing	2.090	2.122	2.223	2.420	3.250
StarNT	2.032	2.072	2.139	2.292	2.736
Previous with changed capital conversion (Section 3.1)	2.018	2.046	2.104	2.277	2.854
Previous with changed dictionary mapping (Section 3.2)	2.010	2.040	2.092	2.272	2.879
Previous with separate mapping (Section 3.3)	1.987	2.013	2.060	2.243	3.047
Previous with new dictionary (Section 3.4)	1.959	1.988	2.019	2.211	2.909
Previous with longer codes (Section 3.4)	1.945	1.980	2.025	2.192	2.786
Previous with changed dictionary ordering (Section 3.5)	1.941	1.977	2.019	2.184	2.784
Previous with added matching of shorter words (Section 3.6)	1.940	1.975	2.018	2.183	2.777
Previous with added q -gram replacement (Section 3.7)	1.935	1.971	2.014	2.172	2.745
Previous with added EOL-coding (WRT) (Section 3.8)	1.894	1.913	1.971	2.131	2.682

StarNT with all the introduced improvements will be called word replacing transformation (WRT).

The experiments are demonstrated on the example of `book1`, a relatively large and typical file from the Calgary corpus¹. The output of StarNT is compressed by best-performing and well-known compressors. PAQ6 [32], PPMonstr [33], and UHBC [34] are currently among the most efficient compressors that do not use preprocessing techniques. UHBC, a BWT-based compressor, was run in its maximum compression mode (`-m3`) with large BWT blocks. The results of widely used `bzip2` [35] and `gzip` [36] are also included. The chosen programs represent all the main approaches to lossless data compression.

3.1 Capital conversion

Capital conversion (CC) is a well-known preprocessing technique [37, 7]. It is based on the observation that words like for example `compression` and `Compression`, are almost equivalent, but general compression models, e.g., plain PPM, cannot immediately recognize the similarity. The CC idea is to convert the capital letter starting a word to its lowercase equivalent and denote the change with a flag. Additionally, it is worth using another flag to mark a conversion of a full uppercase word to its lowercase form. Both forms

¹<http://corpus.canterbury.ac.nz/descriptions#calgary>

are used in StarNT. What we changed here is the position of the flag: in StarNT it was appended to the transformed word, in our algorithm the flag is put in front of the word. Our approach has two advantages: the flag itself is easier to predict as it usually appears as the first non-whitespace character after punctuation marks like period, exclamation mark, etc., and the flag does not hamper the prediction of the following word.

The described CC technique gives even better results if a space is added right after the flag [7], i.e., between the flag and the encoded word. It helps to find a longer context with PPM as well as helps other algorithms, which can be observed in Table 1. Such CC variation is not perfect for LZ77 compressors; a more suitable form will be presented in Section 5.

Abel and Teahan [9] report a slight gain obtained from yet another idea: refraining from the conversion of such words that occur in the text only with the first letter capitalized. The benefit of this idea for compression is obvious but it requires an additional pass over the text, and that is why we decided not to implement it.

3.2 Dictionary mapping

Ordinary text files, at least English ones, consist solely of ASCII symbols not exceeding 127 in total. The higher values are unused, and thus could be spent for the codewords. At the first attempt, we used values from 204 to 255 only. This means 52 symbols, just as in StarNT, but the obvious difference is that in our algorithm the alphabets for codewords and for original words do not intersect. Consequently, flag f_{or} is no longer needed. As can be seen in Table 1, our modification boosts the compression with all chosen compressors except `gzip`. Part of this gain is achieved by eliminating the flag f_{or} , but disjoint subalphabets are important too. We conclude that the StarNT authors were wrong to have claimed higher compression resulting from the codeword alphabet limited to `[a..zA..Z]`.

3.3 Separate alphabets for codewords

Disjoint alphabets for codewords and original words are not actually our own idea, as their benefits were noticed earlier by Smirnov [23]. As the bytes of codewords and original words cannot be confused, PPM and BWCA compressors are more precise in their predictions.

As pointed out in the previous section, we have at least 128 symbols for codewords available. This is a fair amount and thus Smirnov’s idea can be extended using disjoint subalphabets for the successive symbols of the codewords. Specifically, we chose subalphabets of sizes 43 (range 128..170), 43 (range 171..213), and 42 (range 214..255). In this way we can store up to $43 + 43 \times 43 + 43 \times 43 \times 42 = 79,550$ different words, more than enough for the StarNT dictionary, which has about 60,000 words. As Table 1 shows, this modification helps all compressors but `gzip`.

3.4 Dictionary organization

As written in Section 2, words in the StarNT dictionary are sorted according to their length, and within groups of words of equal length, the alphabetical ordering is used. So far we have carried out experiments with the original dictionary used in the work of Sun *et al.* (about 60,000 words), but now we have replaced it with Aspell’s English dictionary level 65 (about 110,000 words) [38], reduced to 80,000 entries by discarding the rarest words. Additionally, we allow up to 4 bytes per codeword, and use the subalphabet sizes $\langle 101, 9, 9, 9 \rangle$ for the four codeword bytes. The choice was motivated experimentally. The codeword bytes are emitted in the reverse order, i.e., the range for the last byte has always 101 values. As can be seen in Table 1, the new dictionary improves the compression performance with all the tested compressors.

3.5 Word order in the dictionary

The most fundamental law of compression says that more frequent messages should be represented with shorter codes than less frequent messages. As the codewords in StarNT are of variable lengths, it is clear that several most popular words should be encoded with a single byte, whereas a number of less frequently used words should be represented by pairs of bytes, etc. From this point, words should be ideally sorted according to their frequency in the given text but storing the words’ frequencies (or relative order) is obviously prohibited, and selecting the codewords adaptively seems difficult. In our dictionary, words are thus sorted with the relation to their frequency in a training corpus of more than 3 GB English text taken from the Project Gutenberg library². The word order is however more complex. We found it of benefit to first sort the dictionary according to the frequency, and then sort in small groups according to the lexicographical order of suffixes, i.e., sorting from right to left. Correspondingly to the chosen parameters presented in Section 3.4, 10 ($= 1 + 9$) the first small groups have 101 items each, 9 subsequent groups have 909 ($= 101 \times 9$) items, and finally all the following ones have 8,181 ($= 101 \times 9 \times 9$) items. Such a solution improves the contextual prediction in the latter phase.

3.6 Matching shorter words

Having separate alphabets for original words and codewords, it is easy to encode only a prefix of a word, if the prefix matches some word in a dictionary but the whole word doesn’t. The original StarNT cannot encode word prefixes as the output codeword and the suffix of the original word would be glued and thus be ambiguous for the decoder. For this modification, StarNT would need an additional flag, e.g., f_{sep} , in order to separate the codeword and the suffix of the original word. Introducing such a flag may not be compensated by the gain achieved from the substitution. Still, even in our case the gain is negligible.

²<http://www.gutenberg.org>

3.7 Q -gram replacement

Substituting frequent sequences of q consecutive characters, i.e., q -grams, with single symbols [19] is a “little brother” of the word-based substitution, and belongs to the most obvious techniques. The underlying idea is, in context-based compressors, to promote faster learning of the contextual statistics, and to virtually increase the compressor’s context order (this does not apply to BWCA), or, in LZ77 compressors, to mainly save bits on sequence offsets and lengths.

The sequence length q usually does not exceed 4. It is natural to represent q -grams with characters hardly ever used in plain texts. In our scheme, however, the values above 127 in ASCII code are already used as the codeword alphabet for the word dictionary. On the other hand, a nice side-effect of our capital conversion variant is that we get rid from all the capital letters, and therefore those 26 values could be spent for encoding q -grams. We make use of this idea.

3.8 End-of-Line coding

End-of-Line (EOL) coding invented by Taylor³ is a preprocessing method founded on the observation that End-of-Line symbols⁴ are “artificial” in the text and thus spoil the prediction of the successive symbols. The general idea is to replace EOL symbols with spaces and to encode information enabling the reverse operation in a separate stream.

This idea has been incorporated in many existing compressors. According to our knowledge, one of its best implementations is used in DC compressor [29] by Binder. EOL symbols are converted to spaces, and for each space in the text, DC writes a binary flag, to distinguish an original EOL from a space. The flags are encoded with an arithmetic coder, on the basis of an artificial context. The information taken into account to create the context involves the preceding symbol, the following symbol, and the distance between the current position and the last EOL symbol. DC appends the EOL stream at the end of the processed file.

We have implemented a variation of DC’s solution. In our implementation, similar to Reference [9], only those EOL symbols that are surrounded by lowercase letters should be converted to spaces. EOL coding slightly deteriorates the compression on the files `progp` and `progc` (as these are program source files and have very irregular line lengths), but gives a huge improvement on `book1` and `book2`.

3.9 Binary data filter

A practical text preprocessor should be protected from a significant loss on data which do not fit its model (in particular, binary data). StarNT, for example, does not meet such a requirement as it plainly encodes special symbols in the input data, e.g., input symbols from the codeword alphabet, with two bytes, the first of which is an escape (flag f_{esc}). As a result, some compression loss can be seen on the binary files from Calgary corpus

³M. Taylor. Private communication, 1998.

⁴CR+LF, LF, or CR.

(geo, obj1, obj2). Our goal was not simply to preserve the compression rate on binary files, but also improve it, if the binary file contains some parts of text (words) inside.

Our solution behaves identical to StarNT if the fraction of non-textual characters in the input stream is low. If, however, at least 20% of characters so far seen are non-textual ones, then sending the flag f_{esc} signals that the r successive characters will be copied verbatim to the output. In other words, f_{esc} is used to mark the beginning of a non-textual data chunk, rather than encoding a single non-textual character as StarNT does. The encoder switches back to the textual regime if r preceding characters are all textual. Experiments have shown that $r = 16$ is the best choice for the Calgary corpus.

3.10 Surrounding words with spaces

Punctuation marks, like periods and commas, usually appear just after a word, without a separating space. It implies that if a given word is sometimes followed by a (for example) comma, and sometimes by a space, the encoding of the first character after this word is relatively expensive. Inserting a single space in front of a punctuation mark relieves this prediction. Of course, the punctuation mark itself has still to be encoded but its context is now more feasible and overall the loss from expanding the text is more than compensated. This minor idea has already been used in a few compressors, e.g., PPMN [28] and Durilca [24].

We modified this technique with the idea of surrounding words with spaces rather than preceding punctuation marks with spaces. Let us call the words which are not followed by a space as badly surrounded. We add two new flags: f_{sb} inserted before a space preceding a badly surrounded word, and f_{sa} inserted after a space following such a word. Of course, it makes a difference only in some cases, like two words separated by a hyphen or two adjacent punctuation marks, e.g., a comma followed by a quotation mark.

This technique provides a benefit only if there are at least a few occurrences of the word, because it joins similar contexts (helps in better prediction of the word's first symbol as well as the next symbol just after the word). For this reason, we decided to employ this idea only for files larger than 1 MB. Please note that such a conversion is never used on the Calgary corpus.

4 Experimental results

For experiments, we have chosen PAQ6 and PPMonstr (var. 1) as they yield the best compression rates among the compressors with no preprocessing or specialized models for text. It guarantees that our transformations do not interfere with possible preprocessing or specialized modeling built in the compressor. PPMonstr was run in order-8 with 256 MB memory limit. PAQ6 was run with the -6 switch. For a reference, we tested also Durilca 0.4 with the following switches: -t2 -m256 -o8. The compression ratios are expressed in bits per character (bpc). All tests were carried out on an Athlon XP 2500+ machine, clocked at 1833 MHz and equipped with 512 MB RAM, under Windows 2000 operating system. WRT was implemented in C++ and built with MinGW 3.4.0 compiler.

The first test was executed on the Calgary corpus and the results are shown in Table 2. For almost all the files, WRT outperforms StarNT. With PPMonstr as the final compres-

Table 2: Compression ratios for the Calgary corpus. (Since StarNT is unable to improve compression of binary files, the files marked with asterisk are compressed without StarNT preprocessor.)

File	PAQ6	StarNT + PAQ6	WRT + PAQ6	PPMonstr	StarNT + PPMonstr	WRT + PPMonstr	Durilca
bib	1.617	1.579	1.520	1.679	1.625	1.570	1.533
book1	2.090	2.032	1.894	2.122	2.072	1.913	1.846
book2	1.691	1.670	1.583	1.754	1.720	1.615	1.640
geo	3.536	3.536*	3.538	3.874	3.874*	3.875	3.783
news	2.034	2.007	1.919	2.104	2.051	1.952	1.951
obj1	3.047	3.047*	3.012	3.353	3.353*	3.305	3.170
obj2	1.703	1.703*	1.717	1.925	1.926*	1.939	1.844
paper1	2.052	1.934	1.829	2.131	2.001	1.914	1.864
paper2	2.056	1.924	1.798	2.119	1.974	1.849	1.761
pic	0.456	0.456*	0.445	0.700	0.700*	0.700	0.506
progc	2.031	1.987	1.911	2.121	2.044	2.039	1.975
progl	1.314	1.276	1.253	1.386	1.328	1.318	1.368
progp	1.312	1.311	1.295	1.427	1.409	1.412	1.408
trans	1.126	1.126*	1.091	1.194	1.194*	1.164	1.073
Average	1.862	1.828	1.772	1.992	1.948	1.898	1.837
Total time [s]	191.2	153.4	146.9	8.2	9.0	7.7	11.4

sor, WRT achieves average gain of 2.6% over StarNT, and 4.7% over the unprocessed files. WRT’s gain with PAQ6 over StarNT is 3.1%, while 4.8% over the unprocessed files. The biggest gain is achieved on **book1** (resulting mainly from the EOL-encoding), as well as on **paper1** and **paper2**. The compression of **geo** and **pic** is practically unaffected by WRT, because those files do not contain any textual data. There are three binary files in the corpus that do contain some textual data: **obj1**, **obj2**, and **trans**. WRT yields gain on **obj1** and **trans**, but slightly deteriorates compression on **obj2** due to unnecessary EOL-coding.

Overall, WRT+PAQ6 wins in our “top compression” competition. But PPMonstr is at least an order of magnitude faster than PAQ6 and needs a moderate amount of memory to work, in contrast to PAQ6’s constant (and high) memory requirements.

For the next experiment we took nine large plain text files, mostly containing classic English literature (for example, King James Bible, “Alice’s Adventure in Wonderland”, and a collection of Dickens’ writings). Their sizes vary from 122 KB⁵ (**asyoulik**) to about 10 MB (**dickens**). The files come from both Canterbury corpora⁶ (**alice29**, **asyoulik**, **lcet10**,

⁵Throughout the paper, by 1 KB we mean 1,024 bytes.

⁶<http://corpus.canterbury.ac.nz/descriptions/#large>

Table 3: Compression ratios for the large files

File	PAQ6	StarNT + PAQ6	WRT + PAQ6	PPMonstr	StarNT + PPMonstr	WRT + PPMonstr	Durilca
1musk10	1.718	1.676	1.567	1.743	1.701	1.568	1.489
alice29	1.927	1.841	1.707	1.979	1.868	1.731	1.571
anne11	1.909	1.846	1.724	1.942	1.875	1.728	1.642
asyoulik	2.176	2.080	1.970	2.243	2.120	2.018	1.918
bible	1.314	1.294	1.263	1.369	1.315	1.280	1.259
dickens	1.705	1.676	1.607	1.715	1.691	1.614	1.594
lcet10	1.671	1.625	1.514	1.732	1.675	1.537	1.482
plravn12	2.104	2.044	1.965	2.141	2.090	1.978	1.965
world95	1.132	1.109	1.105	1.190	1.134	1.139	1.080
Average	1.740	1.688	1.602	1.784	1.719	1.621	1.556
Total time [s]	1285.3	848.5	862.2	45.0	43.2	41.9	60.2

plravn12, bible), Archive Comparison Test web site⁷ (1musk10, alice29, world95) and the Silesia corpus⁸ (dickens). As Table 3 shows, text preprocessing always helps on such data.

The best results were obtained by Durilca. Its advantage over WRT+PPMonstr reaches 3.2% on the Calgary corpus and 4.0% on the large files. Apart from the record-breaking compression ratios achieved at a relatively good speed, also the elegance of the concept may be seen as its great merit. Although the excellence of Shkarin’s work cannot be denied, we would like to point out a few weak points of this approach. Firstly, using a built-in model is specific for PPM compression (and, perhaps, LZ78). Secondly, although in principle the idea may be applicable to most modern PPM implementations, it is so far used only with the compressors Durilca and Durilca Light, which are based on Shkarin’s PPMonstr and PPMd, respectively (this might change in the future). On the other hand, WRT (and StarNT, assuming the implementation were public) is ready for immediate use with any compressor. Thirdly, Durilca’s model for English text, in the PPMonstr format, takes 563 KB⁹ in an external file, while the standard WRT dictionary takes 151 KB in a properly compressed form. Fourthly, Durilca does not constrict the input data before PPM encoding, which means that in very low order (e.g., 2) the compression ratio is rather mediocre, and clearly worse than that of WRT followed by PPMonstr working at the same context order. It should be pointed out that the compression difference between order-2 and order-8 PPMonstr is only 4–7% on WRT output (for order-2, LZ77-oriented

<http://corpus.canterbury.ac.nz/descriptions/#cantrbry>

⁷<http://compression.ca>

⁸<http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm>

⁹This refers to the previous version, v0.3a, as in the latest v0.4 all the used models are kept in a single file. Still, supposedly, the English text model description in the current version does not take less space.

Table 4: Summary results for the Calgary corpus

Preprocessing method	gzip	UHARC	bzip2	UHBC	PPMd	PPMonstr	PAQ6	RKC	Durilca
none	2.695	2.354	2.368	2.208	2.114	1.992	1.862	1.903	1.837
StarNT	2.537	2.226	2.297	2.149	2.070	1.948	1.828		
WRT	2.526	2.179	2.225	2.061	2.012	1.898	1.772		
(imp. on none)	(6.27%)	(7.43%)	(6.04%)	(6.66%)	(4.82%)	(4.72%)	(4.83%)		
(imp. on StarNT)	(0.43%)	(2.11%)	(3.13%)	(4.09%)	(2.08%)	(2.57%)	(3.06%)		
WRT-LZ77	2.429	2.180	2.290	2.127	2.057	1.949	1.857		
(imp. on none)	(9.87%)	(7.39%)	(3.29%)	(3.67%)	(2.70%)	(2.16%)	(0.27%)		
(imp. on StarNT)	(4.26%)	(2.07%)	(0.30%)	(1.02%)	(0.63%)	(-0.05%)	(-1.59%)		

variant of WRT used, see Section 5), which means that very low orders may yield a practical trade-off, remembering the big difference in speed and memory use. To sum up the drawbacks: in some aspects, Durilca is much less flexible than dictionary-based text compressors.

Our preprocessor boosts the compression of the leading PPM algorithms by nearly 5% on the Calgary corpus and even 8–9% on the large files. Those results are by 2–6% better than the respective ones achieved with the aid of StarNT. Tables 4 and 5 present the summary results. For this test, we enlarged the list of used compressors. The LZ77 family is represented by the widely used `gzip` and `UHARC 0.4` by Herklotz [39] with switches `-m3 -mm- -md16384`; the latter is a modern variant with arithmetic coding and 16 MB sliding window which performed best with WRT among the LZ77 compressors. There are two compressors from the BWT-based family, the popular `bzip2` and `UHBC 1.0` by Herklotz, currently the leader in its category. Finally, we added `PPMd var. 1` by Shkarin and `RKC 1.02` by Taylor; the latter uses its own text preprocessing, including a static dictionary. `PPMd` worked in order-8 with 256 MB of memory. `RKC` also used 256 MB for the model and was run with the order 8. The full syntax for `RKC`, with all text filters set on, was: `-M256m -o8 -B16m -mxx -te+ -ts+ -td+ -a+ -Lenglish`.

As can be seen, the gain from WRT preprocessing is often in the order of 10% or even more, and it clearly outperforms StarNT, sometimes by even more than 6%. On the other hand, `RKC`, incorporating similar ideas to ours, stays close to the results of `WRT+PPMonstr`, losing 2.1% on the large files and only 0.3% on the Calgary corpus. Additional experiments suggest that the used EOL coding variant is `RKC`'s forte. Unfortunately, Taylor's compressor is not available with sources.

Regarding the speed (see Tables 2 and 3), WRT makes `PAQ6` work up to 1.5 times faster, while with `PPMonstr` the improvement does not exceed a few percent. On the other hand, although our WRT implementation is faster than StarNT, and produces

Table 5: Summary results for the large files

Preprocessing method	gzip	UHARC	bzip2	UHBC	PPMd	PPMonstr	PAQ6	RKC	Durilca
none	2.833	2.329	2.105	1.890	1.844	1.784	1.740	1.656	1.556
StarNT	2.401	2.045	1.976	1.814	1.773	1.719	1.688		
WRT	2.392	1.913	1.851	1.694	1.665	1.621	1.602		
(imp. on none)	(15.57 %)	(17.86 %)	(12.07 %)	(10.37 %)	(9.71 %)	(9.14 %)	(7.93 %)		
(imp. on StarNT)	(0.37 %)	(6.45 %)	(6.33 %)	(6.62 %)	(6.09 %)	(5.70 %)	(5.09 %)		
WRT-LZ77	2.059	1.830	1.806	1.730	1.692	1.658	1.657		
(imp. on none)	(27.32 %)	(21.43 %)	(14.20 %)	(8.47 %)	(8.24 %)	(7.06 %)	(4.77 %)		
(imp. on StarNT)	(14.24 %)	(10.51 %)	(8.60 %)	(4.63 %)	(4.57 %)	(3.55 %)	(1.84 %)		

more compact outputs, it does not always imply faster overall performance with PPM-like compressors. This is related to the well-known phenomenon of PPM’s significant degradation in speed on more “dense” data. Still, the differences are rather slight and in most cases WRT wins. It should also be noticed that Durilca is by 30% to 50% slower than PPMonstr working with the same context order.

We defer commenting the last rows in Tables 4 and 5 until Section 5, where an LZ77-specific variant of WRT will be presented.

5 Lempel–Ziv compression optimized modeling

Compressors from the Lempel–Ziv’77 family differ significantly from PPM and BWCA. Basically, they parse the input data into a stream of matching sequences and single characters. The LZ matches are encoded by specifying their offsets, i.e., distances to the previous sequence occurrence in a limited past buffer, and lengths, while the literals are encoded directly, in most variants with Huffman coding. There have been many refinements since the seminal works of Ziv and Lempel [40] and Storer and Szymanski [41], but the overall picture has not changed. LZ77 compressors do not predict characters on their context basis, or, in some modern variations, they do it only in a very low order rudimentary manner. The strength of LZ77 lies in succinct encoding of long matching sequences but in practice most matches are rather short, and again the plain encoding scheme cannot compete with modern compressors, although it enables high decoding speed. In consequence, a text preprocessor adapted for LZ77 compressors should attempt to:

- reduce the number of single characters to encode;
- decrease the offset (in bytes) of the matching sequences;

- decrease the length (in bytes) of the matching sequence;
- virtually increase the sliding window, i.e., the past buffer from which matching sequences are found.

Those considerations lead to the (experimentally validated) conclusion that almost any idea to shorten the output of the preprocessor is also beneficial for the final compression. This was not the case with PPM or BWCA.

In accordance with this observation, we extended the alphabet for codewords from 128 to 189 symbols ($[A..Z]$, 0..8, 14..31, and others). Another important change is elimination of most spaces between words. Since usually a word is followed by a single space, only exceptions from this rule require special treatment. Such an assumption is known as the spaceless word model [42]. Still, without spaces between words, there must be a way to detect codeword boundaries. Tagged Huffman and similar schemes for compression and search on words distinguish between the first (or the last) byte of a wordcode and the other bytes. Traditionally, this has been performed by sacrificing one bit per byte but recently more efficient schemes [43, 44] have been presented. Our scheme is analogous to the one from the mentioned work in specifying unequal ranges for the first and the latter codeword bytes. We have run the algorithm for obtaining the optimal (s, c) -dense code for a given distribution from Reference [44], and it suggested the alphabet partition $\langle 135, 54, 54 \rangle$ for our training corpus. We, however, use the partition $\langle 167, 22, 22 \rangle$, chosen experimentally. The codewords have at most 3 bytes, the second and third bytes use the same alphabet (cf. Section 3.4). The difference between the theoretical and experimentally found partition probably means that the distribution of words in the training and test data do not fit perfectly.

Other modifications are of lesser importance. The ideas of surrounding words with spaces and q -gram replacement are no longer used. The dictionary is augmented with several most popular 2-letter English words, and the dictionary items are now sorted merely by their frequency in training data. The EOL coding is simplified: all EOL symbols are converted to spaces. Finally, capital conversion is changed in two details: (i) before the actual conversion the case of the first letter after a period, quotation, or exclamation mark, is inverted, which results in fewer CC flags, and (ii) no space is added just after CC flags.

Tables 4 and 5 show that the proposed preprocessing variant is significantly better for LZ77 compressors, especially `gzip`. On the large files, the improvement for `gzip` in comparison to the original WRT reaches almost 14%, while with UHARC it is 4.3%. On the Calgary corpus files the gains vary but on average are only moderate (even a tiny average loss with UHARC, comparing to the original WRT). It is interesting to note that the modified preprocessing is more suitable for `bzip2` on the large files. This does not happen with the more advanced BWT-based compressor, UHBC. Detailed results of LZ77 compressors after various preprocessing variants are presented in Table 6.

6 Dictionary search

The most serious disadvantage of dictionary-based text preprocessors (including WRT) is the necessity of keeping quite a large static collection of words. It affects both the

Table 6: Compression ratios of Lempel–Ziv methods with and without preprocessing

File	gzip	StarNT +gzip	WRT +gzip	WRT-LZ77 +gzip	UHARC	StarNT +UHARC	WRT +UHARC	WRT-LZ77 +UHARC
1musk10	2.906	2.423	2.383	2.005	2.356	2.067	1.908	1.794
alice29	2.850	2.405	2.330	2.044	2.527	2.149	2.023	1.923
anne11	3.025	2.547	2.474	2.099	2.539	2.204	2.031	1.923
asyoulik	3.120	2.694	2.707	2.374	2.782	2.387	2.275	2.231
bible	2.326	2.006	2.026	1.744	1.853	1.675	1.624	1.500
dickens	3.023	2.495	2.525	2.098	2.334	2.076	1.954	1.829
lcet10	2.707	2.255	2.184	1.902	2.256	1.972	1.788	1.718
plrabn12	3.225	2.727	2.770	2.371	2.750	2.399	2.187	2.140
world95	2.311	2.053	2.126	1.894	1.564	1.476	1.431	1.415
Average	2.833	2.401	2.392	2.059	2.329	2.045	1.913	1.830

memory usage and the size of the preprocessing package. Therefore it is crucial to keep the dictionary possibly tight but also in a way beneficial for the filtering speed. There are many data structures which can be used for storing the dictionary but requirements for both speed and memory efficiency rule out some of them.

The authors of StarNT employ a ternary search tree (TST) for maintaining the dictionary. We evaluate competitive data structures, from which the most promising ones are those based on hashing. The experimental results for all the implemented methods are presented in Table 7. A concatenation of all the test files used in our experiments, containing about 3 million words, was used for comparison. For each data structure, we measured the average time of returning a code for a word. In WRT, we allowed not only full-word matches in the dictionary, but also partial ones, i.e., finding a word being the longest prefix of the examined word. Therefore, the results both for only full-word and also partial matches are presented. We also show the memory requirements of the data structures.

The first competitor we examined was chained hashing. Its advantages are simplicity and moderate memory usage. The main disadvantage is a need to handle collisions, which slows down the preprocessing. To minimize this effect we can enlarge the hash table sacrificing memory. Several results for hash tables of sizes $x \times n$, where $x = 1, 4, 8$, are given in rows denoted by HASH-CHAINED- x , and n is the number of entries (words) in the dictionary. This method is rather slow for small values of x , but for $x = 8$ is competitive with TST.

A large hash table with open addressing can reduce the number of collisions also. We examined the hash table of a size about $25n$ and a double hashing function carefully selected to guarantee not more than 3 executions of a hash function per word. This method (HASH-OPEN) is faster than TST, but requires more memory. The necessity of finding the hash functions specific for the given dictionary is also a serious drawback.

Table 7: Word search time and memory usage for various dictionary representations

Method	Memory [KB]	Avg. search time [ns/word]	Avg. search time (also partial words) [ns/word]
TST	7,199	158	168
HASH-CHAINED-1	2,902	259	278
HASH-CHAINED-4	3,839	168	182
HASH-CHAINED-8	5,089	153	163
HASH-OPEN	10,579	134	144
MPH	1,029	115	120

A method for solving the collision problem definitely is perfect hashing [45]. A minimal perfect hashing function can be constructed rapidly with several algorithms. One is an application of minimal finite state automata (MFSA). MFSA building is fast (linear in total length of all words from the dictionary) and needs little memory [47, 48]. The automaton stores also the whole dictionary and therefore the explicit word list is unnecessary. After MFSA is built, the hash function value for a word can be determined in time linear in the word length [46]. The results for this method are presented in the table in row MPH.

As Table 7 shows, the methods HASH-CHAINED-8, HASH-OPEN, and TST are comparable in execution speed and memory usage. The solution based on minimal perfect hashing outperforms the others by about 20% in speed and drastically in memory requirements.

7 Implementation details

The routines for making and searching the MFSA are based on the source code accompanying Reference [48]. The algorithm of calculating the MPH value is a modification of the method presented by Lucchesi and Kowaltowski [46]. They show how the unique value for each word can be obtained by summing up the values related with states during the traversal of the automaton. Each state stores the number of different routes from it to any final transition. The sample automaton presenting this idea is shown in Figure 1a. To obtain a hash index one needs to initialize the counter with a value 0 and increase it every time it moves through a terminal transition that does not finish the examined word. The counter should be also increased in states by all the values pointed by the transitions denoted by lexicographically lower attributes than the current one.

For speed reasons, we postulate to accompany each transition, instead of states, with a value being the number of different routes to any final transition outgoing from the same state as the current transition, whose attributes (characters) are lower than the current one. This method raises the total memory requirements by about 30% but reduces the

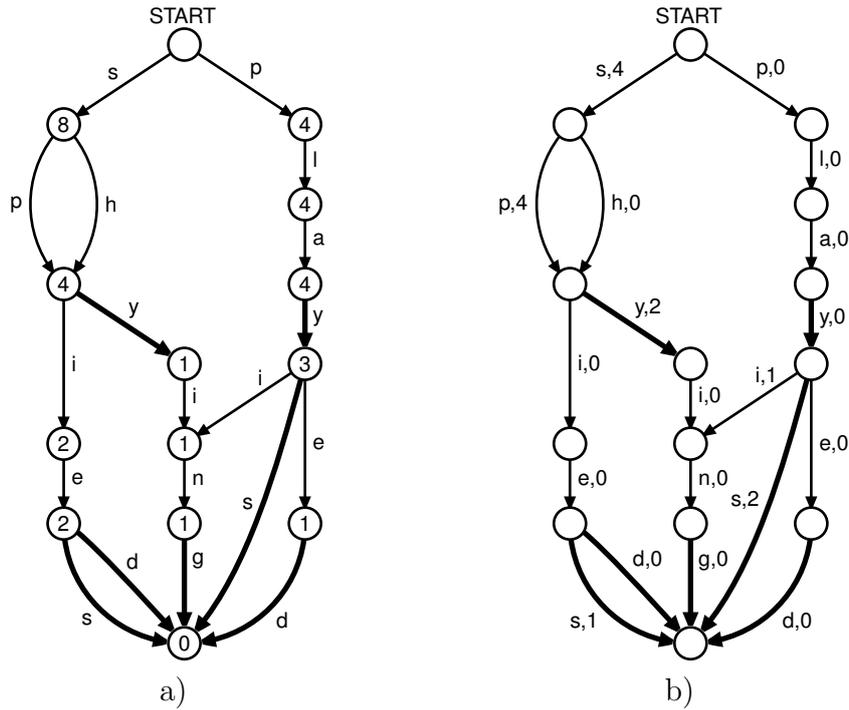


Figure 1: Sample automata allowing calculation of minimal perfect hashing values for a set of words: {play, played, playing, plays, shied, shies, shy, shying, spied, spies, spy, spying}. Two variants are shown: a) values are associated with states; b) values are associated with transitions.

number of considered values, because one only needs to sum up the values related with the traversed transitions to obtain a hash index. Sample MFSA illustrating this modification is presented in Figure 1b.

Another asset of this technique is the possibility of dynamically changing the order of transitions within the state to reduce the number of transitions searched before the one for the examined word character is found. To this end, we always move the transition that is traversed one position to the front of the list of all state transitions. In this way, the time necessary to calculate MPH value is reduced because the most common routes in the automaton are fast to find.

The source codes of WRT and WRT-LZ77 are available on-line [49, 50].

8 Limited dictionary scenario

Succinct representation of the dictionary actually means two related but different things. One is to use as little memory as possible during the processing (and if possible, without compromising the speed), the other is to keep the dictionary on disk in a possibly compact archive. The latter goal is crucial if many dictionaries (e.g., for different languages) are distributed with an archiving software. A trivial way to minimize both the operating memory use and disk storage is to reduce the number of entries in the dictionary. And again, the most trivial way of pruning the dictionary is to discard the rarest words, i.e.,

to truncate the dictionary after m first entries, $m < n$, where n is count of words in the dictionary. Obviously, if m is much smaller than n , a noticeable loss in compression must occur. We examined the effect of truncating the dictionary after the first 25%, 50%, and 75% words on the compression ratios of our test files and on the size of the dictionary itself, compressed with `gzip -9` and `PPMonstr`.

As Table 8 shows, a large dictionary is most important for LZ77 compressors, especially the weaker of the tested pair, `gzip`. Truncating the dictionary to 20,000 words results in 2.7% compression deterioration with `gzip` while with PPM and BWCA compressors the loss does not exceed about 1.5%. The best compromise in many cases may be halving the dictionary, i.e., reducing it to 40,000 words, as the compression loss is bounded by about 0.4% (only with `gzip` it is 0.7%).

As said, another practical issue is compact representation of the dictionary for distribution purposes. The lexicographical order of words would be most feasible for the compression. Unfortunately, we cannot change freely the order of words, as otherwise some auxiliary data would have to be added to reconstruct the dictionary in the form designed for WRT. Still, there exists a partial order in our dictionary, as words are sorted in groups according to their suffixes, as described in Section 3.5. `PPMonstr` run in order-5 (most successful here) on the raw dictionary of 80,000 items, with words separated with LF End-of-Line characters, resulted in 200.8 KB. A folklore solution for improving the compression on sorted sets of strings (sometimes called front compression) is to make use of the prefix shared by two successive entries. This however, adapted to common suffixes, appeared only a moderate success with our dictionary, for example, the improvement with `PPMonstr -o5` reached 6.2%. Even plain reversing each word fared better: 16.0% gain with order-6 `PPMonstr`. It is understandable as common word prefixes all appear in the fixed order-1 context (the EOL character). Using 2-byte EOL's prolongs the context and gives further gain but it is not a breakthrough. Much more successful was the "generalizing" idea suggested by Shelwien [51]. He proposed, assuming a lexicographically sorted dictionary, to follow each word with a number (equal for each line) of LF EOL symbols and use PPM with high enough context orders, so as to alleviate the prediction of the first letters in a word while keeping accurate prediction on the farther letters. In our variation of Shelwien's idea, words were first reversed, so as to have common prefixes, and then each word was followed by 14 spaces and one LF+CR EOL character. In this way, `PPMonstr` (in order-16) achieved 151.0 KB, that is 24.8% less than on the original representation. Using a specialized compressor for a dictionary may bring an additional gain of a few percent [51] but we have refrained from such experiments.

Squeezing the dictionary with `gzip` is also quite interesting from a practical point. On the raw dictionary it gets 293.1 KB, while the most appropriate preprocessing technique here seems to be front compression on the reversed words (removing common prefixes of words and sending the prefix lengths to a separate stream): obtained 231.2 KB is about 21.1% less than with no preprocessing. We have tried out a few modifications of those simple solutions (both for `PPMonstr` and `gzip`) but we could not improve the results.

Compacting a truncated dictionary results in more or less proportional savings. For example, `PPMonstr` can squeeze the dictionary with 40,000 words to 72.7 KB, and the smallest one to 37.3 KB.

Table 8: Compression ratios for the large files with varying dictionary sizes. For `gzip` and `UHARC`, the LZ77-specialized WRT version was used.

Dictionary size	<code>gzip</code>	<code>UHARC</code>	<code>bzip2</code>	<code>UHBC</code>	<code>PPMd</code>	<code>PPMonstr</code>	<code>PAQ6</code>
80,000	2.059	1.830	1.851	1.694	1.665	1.621	1.602
60,000 (det. to full dict.)	2.062 (0.15 %)	1.832 (0.11 %)	1.853 (0.11 %)	1.696 (0.12 %)	1.666 (0.06 %)	1.622 (0.06 %)	1.604 (0.12 %)
40,000 (det. to full dict.)	2.073 (0.68 %)	1.837 (0.38 %)	1.859 (0.43 %)	1.701 (0.41 %)	1.670 (0.30 %)	1.627 (0.37 %)	1.608 (0.37 %)
20,000 (det. to full dict.)	2.115 (2.72 %)	1.858 (1.53 %)	1.879 (1.51 %)	1.719 (1.48 %)	1.684 (1.14 %)	1.642 (1.30 %)	1.623 (1.31 %)
0 (det. to full dict.)	2.768 (34.43 %)	2.284 (24.81 %)	2.056 (11.08 %)	1.843 (8.80 %)	1.780 (6.91 %)	1.729 (6.66 %)	1.723 (7.55 %)

9 Conclusions and future work

We presented a static dictionary-based preprocessor, called word replacing transformation (WRT), intended for text compression. Our ideas are significant extensions of the mechanisms present in former algorithms, like `StarNT` and `LIPT`. WRT distinguishes favorably from its competitors both in the performance of the subsequent compression and in speed. We proposed two variants of the transform: one suitable for PPM and BWCA compressors, and one tailored for LZ77 compressors. The output of WRT-LZ77 is more compact and sometimes comparable in size to the output of `gzip` run on the raw text, which means that the LZ77-oriented version of WRT can be perceived as a separate, and very fast, specialized compressor.

On plain English text, WRT improves the compression performance of `bzip2`, a popular BWT-based compressor, by about 12%, with order-8 `PPMd` the gain is about 9% on average. Even for the top compressors nowadays, `PPMonstr` and `PAQ6`, the gains are significant: about 9% and 7%, respectively. LZ77 compression can be improved to a higher degree: more than 25% gain with `gzip` has been observed, resulting in compressed outputs often comparable to those produced by `bzip2` from the raw text.

We also discussed the dictionary representation issues, both in main memory “ready for work” and on disk. We found that a dictionary represented in a minimal finite state automaton surpasses chained hash and ternary search tree based solutions both in speed and (several times) in memory occupancy. As for the representation on disk, convenient for software distribution, following each word with a number of spaces and running `PPMonstr` (well adapted to non-stationary data) with a high order appeared the best solution, outperforming variations of the well-known front compression. Additionally,

we examined the trade-off between dictionary size (in raw and compact form) and the compression effectiveness. It appears that a dictionary of 40,000 English words yields a compression loss of about 0.4% only compared to 80,000 words.

An interesting possibility is searching for a word-based pattern, or multiple patterns at once, directly in the WRT-transformed text, preferably in the WRT-LZ77 version (higher data reduction rate, which also implies faster search). Putting apart the symbols not belonging to encoded words, i.e., punctuation marks, letters in words from outside the dictionary, etc., the remaining WRT-LZ77-transformed data can be seen as a sequence of (s, c) -dense code, which is a prefix byte-aligned code enabling fast Boyer-Moore search over it. To keep things simple for any given pattern, the conversion of End-of-Line symbols should be omitted. An additional option is to search directly in the WRT-LZ77 sequence compressed by LZ77 or LZ78 algorithm, which can be seen as a particular application of the algorithms of Navarro and Raffinot [52]. This case, however, seems interesting mainly from the theoretical point, as plain LZ77 and LZ78 compression cannot be particularly efficient on WRT-LZ77 sequence, while the search speedup over the naive “first decompress, then search” approach is only moderate, if any. Careful examination of these possibilities is beyond the scope of this paper.

Several ideas require further investigation. Firstly, words in the dictionary may be labeled with part-of-speech and/or category tags, and Smirnov’s experiments [23] showed that such additional information in the dictionary is beneficial for compression. The drawback of this approach, though, is that preparing dictionaries for various languages gets very laborious. Secondly, it might be possible to achieve slightly higher compression by extending the dictionary with a number of frequent phrases in a given language, consisting of a few words. Our preliminary experiments were unsuccessful though, and this approach complicates the search directly in the transformed text. Thirdly, it is not obvious whether all the presented ideas help for languages other than English. Therefore it would be interesting to gather dictionaries for these languages and observe the impact of the presented ideas applied separately, similar to what has been done in experiments mentioned in Section 3. Finally, for multi-lingual text compression software, succinct representation of dictionaries is crucial and remains a true challenge.

Acknowledgements

The authors would like to thank Maxim Smirnov, Uwe Herklotz, and Jakub Swacha for suggestions of possible improvements, and Weifeng Sun for the source code of StarNT.

References

- [1] Cleary JG, Witten IH. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 1984; **COM-32**(4):396–402.
- [2] Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. *Digital Equipment Corporation (SRC Research Report 124)* 1994. <ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip>.

- [3] Shkarin D. PPM: one step to practicality. In Storer JA, Cohn M, editors, *Proceedings of the 2002 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2002; 202–211.
- [4] Arnavut Z, Magliveras SS. Block sorting and compression. In Storer JA, Cohn M, editors, *Proceedings of the 1997 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1997; 181–190.
- [5] Deorowicz S. Improvements to Burrows–Wheeler compression algorithm. *Software—Practice and Experience* 2000; **30**(13):1465–1483.
- [6] Awan F, Zhang N, Motgi N, Iqbal R, Mukherjee A. LIPT: A Reversible Lossless Text Transform to Improve Compression Performance. In Storer JA, Cohn M, editors, *Proceedings of the 2001 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2001; 481. (Also available at <http://vlsi.cs.ucf.edu/datacomp/papers/dcc2001.pdf>.)
- [7] Grabowski Sz. Text Preprocessing for Burrows–Wheeler Block-Sorting Compression. In *Proceedings of VII Konferencja Sieci i Systemy Informatyczne — Teoria, Projekty, Wdrożenia*, Łódź, Poland, 1999; 229–239. (Also available at http://www.compression.ru/download/articles/text/grabowski_1999_preproc.pdf.rar.)
- [8] Sun W, Mukherjee A, Zhang N. A Dictionary-based Multi-Corpora Text Compression System. In Storer JA, Cohn M, editors, *Proceedings of the 2003 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2003; 448. (Also available at <http://vlsi.cs.ucf.edu/datacomp/papers/dcc2003sun.pdf>.)
- [9] Abel J, Teahan W. Text preprocessing for data compression. Submitted for publication, 2003. http://www.data-compression.info/JuergenAbel/Preprints/Preprint_Text_Preprocessing.pdf.
- [10] Cheney J. Compressing XML with multiplexed hierarchical models. In Storer JA, Cohn M, editors, *Proceedings of the 2001 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2001; 163–172.
- [11] Adiego J, Navarro G, de la Fuente P. Lempel–Ziv Compression of Structured Text. In Storer JA, Cohn M, editors, *Proceedings of the 2004 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2004; 112–121.
- [12] Drinić M, Kirovski D. PPMexe: PPM for Compressing Software. In Storer JA, Cohn M, editors, *Proceedings of the 2002 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2002; 192–201.
- [13] Vo BD, Vo K-P. Using Column Dependency to Compress Tables. In Storer JA, Cohn M, editors, *Proceedings of the 2004 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2004; 92–101.

- [14] Abel J. Record preprocessing for data compression. In Storer JA, Cohn M, editors, *Proceedings of the 2004 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2004; 521. (Also available at http://www.data-compression.info/JuergenAbel/Preprints/Preprint_Record_Preprocessing.pdf.)
- [15] Moffat A. Word based text compression. *Software—Practice and Experience* 1989; **19**(2):185–198.
- [16] Horspool N, Cormack G. Constructing Word-Based Text Compression Algorithms. In Storer JA, Cohn M, editors, *Proceedings of the 1992 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1992; 62–71.
- [17] Dvorský J, Pokorný J, Snásel V. Word-based compression methods and indexing for text retrieval systems. *Lecture Notes in Computer Science* 1999; **1691**:75–84.
- [18] Teahan W, Cleary J. Models of English text. In Storer JA, Cohn M, editors, *Proceedings of the 1997 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1997; 12–21.
- [19] Teahan W. Modelling English text. *Ph.D. dissertation*, Department of Computer Science, University of Waikato, New Zealand, 1998.
- [20] Isal RYK, Moffat A, Ngai ACH. Enhanced Word-Based Block-Sorting Text Compression. In Oudshoorn M editor, *Proceedings of the 25th Australian Computer Science Conference*, Melbourne, January 2002; 129–138.
- [21] Teahan W, Cleary J. The entropy of English using PPM-based models. In Storer JA, Cohn M, editors, *Proceedings of the 1997 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1996; 53–62.
- [22] Kruse H, Mukherjee A. Preprocessing Text to Improve Compression Ratios. In Storer JA, Cohn M, editors, *Proceedings of the 1998 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1998; 556. (Also available at <http://vlsi.cs.ucf.edu/datacomp/papers/dcc97.ps>.)
- [23] Smirnov MA. Techniques to enhance compression of texts on natural languages for lossless data compression methods. *Proceedings of V session of post-graduate students and young scientists of St. Petersburg*, State University of Aerospace Instrumentation, Saint-Petersburg, Russia, 2002. (In Russian.) (Also available at http://www.compression.ru/download/articles/text/smirnov_2002_pos_tagging/smirnov_2002_pos_tagging.pdf.)
- [24] Shkarin D. The Durilca and Durilca Light 0.4a programs. 2004. <http://www.compression.ru/ds/durilca.rar>.
- [25] Chapin B, Tate SR. Higher Compression from the Burrows–Wheeler Transform by Modified Sorting. In Storer JA, Cohn M, editors, *Proceedings of the 1998 IEEE*

- Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1998; 532. (Also available at <http://www.cs.unt.edu/~srt/papers/bwtsort.pdf>.)
- [26] Seward J. The bzip 0.21 program. 1996. <ftp://ftp.elf.stuba.sk/pub/pc/pack/bzip021.zip>.
 - [27] Taylor M. The RKC v1.02 program. 2003. <http://www.msoftware.co.nz>.
 - [28] Smirnov M. The PPMN v1.00b1+ program. 2002. <http://www.compression.ru/ms/ppmb1+.rar>.
 - [29] Binder E. The DC archiver v0.98b. 2000. <ftp://ftp.elf.stuba.sk/pub/pc/pack/dc124.zip>.
 - [30] Gringeler Y. The Compressia 1.0 Beta 1 program. 2003. http://www.softbasket.com/download/s_1644.shtml.
 - [31] Kadach AV. Effective algorithms of lossless text compression. *Ph.D. dissertation*, Russian Academy of Sciences, Institute of Information Systems, Novosibirsk, Russia, 1997. (In Russian.)
 - [32] Mahoney MV. The PAQ6 data compression program. 2004. <http://www.cs.fit.edu/~mmahoney/compression/paq6v2.exe>.
 - [33] Shkarin D. The PPMd and PPMonstr, var. 1 programs. 2002. <http://www.compression.ru/ds/ppmdi1.rar>.
 - [34] Herklotz U. The UHBC 1.0 program. 2003. <ftp://ftp.elf.stuba.sk/pub/pc/pack/uhbc10.zip>.
 - [35] Seward J. The bzip2 1.0.2 program. 2002. <http://sources.redhat.com/bzip2/>.
 - [36] Gailly J-L. The GNU zip (gzip 1.3.5) documentation and sources. 2003. <http://gnuwin32.sourceforge.net/packages/gzip.htm>.
 - [37] Franceschini R, Mukherjee A. Data compression using encrypted text. In Storer JA, Cohn M, editors, *Proceedings of the 1996 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1996; 437. (Also available at <http://csdl.computer.org/dl/proceedings/ad1/1996/7402/00/74020130.pdf>.)
 - [38] Atkinson K. Spell Checking Oriented Word Lists (SCOWL) Revision 5. 2004. <http://wordlist.sourceforge.net/>.
 - [39] Herklotz U. The UHARC 0.4 program. 2001. <ftp://ftp.elf.stuba.sk/pub/pc/pack/uharc04.zip>.
 - [40] Ziv J, Lempel A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 1977; **IT-23**:337–342.

- [41] Storer J, Szymanski TG. Data compression via textual substitution. *Journal of the ACM* 1982; **29**:928–951.
- [42] Moura ES, Navarro G, Ziviani N. Indexing Compressed Text. In Baeza-Yates R editor, *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, Valparaiso, Carleton University Press, 1997; 95–111.
- [43] Rautio J, Tanninen J, Tarhio J. String matching with stopper compression. In Apostolico A, Takeda M, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, Springer, 2002; 42–52.
- [44] Brisaboa N, Fariña A, Navarro G, Esteller M. (S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases. *Lecture Notes in Computer Science* 2003; **2857**:122–136.
- [45] Czech ZJ, Havas G, Majewski BS. Perfect Hashing. *Theoretical Computer Science* 1997; **182**:1–143.
- [46] Lucchesi CL, Kowaltowski T. Applications of Finite Automata Representing Large Vocabularies. *Software—Practice and Experience* 1993; **23**(1):15–30.
- [47] Daciuk J, Mihov S, Watson BW, Watson R. Incremental Construction of Minimal Acyclic Finite State Automata. *Computational Linguistics* 2000; **26**(1):3–16.
- [48] Ciura MG, Deorowicz S. How to squeeze a lexicon. *Software—Practice and Experience* 2001; **31**(11):1077–1090.
- [49] Skibiński P, Deorowicz S. The WRT source codes. 2004. <http://www.ii.uni.wroc.pl/~inikep/research/WRT30d.zip>.
- [50] Skibiński P, Deorowicz S. The WRT-LZ77 source codes. 2004. <http://www.ii.uni.wroc.pl/~inikep/research/WRT-LZ77.zip>.
- [51] Shelwien E. Compressing a dictionary. (In Russian.) 2003. <http://www.compression.ru/sh/2003-08-20.htm>. <http://www.compression.ru/sh/2003-10-24.htm>.
- [52] Navarro G, Raffinot M. A General Practical Approach to Pattern Matching over Ziv–Lempel Compressed Text. *Lecture Notes in Computer Science* 1999; **1645**:14–36.