AUTOMATYKA • 2008 • Tom 12 • Zeszyt 3

Szymon Grabowski*

Making Dense Codes Even Denser

1. Introduction

The task of compressed pattern matching is to report all the occurrences of a given pattern P in a text T available in compressed form. Certain compression algorithms allow for searching without prior decoding, which is practical if the search is fast enough. Some compression algorithms can be adapted easier to the search scenario, while dealing with others is mainly a theoretical challenge. One of the most successful approaches is using a byte code providing compression for a text composed of words.

It is worth to realize at the start that most natural language texts are composed of words and separators between them, so a basic question is how to efficiently handle both words and separators. It is possible to use disjoint alphabets and unrelentingly switch between them. A better practical choice is however the method called spaceless words [17], which assumes that most words are followed by a single space. If this is the case, just the word is encoded. If not (for example, the separator is a comma followed by a space), then the separator is encoded just after the word. At the decoding time, the omitted spaces are inserted after reconstructed words, unless the next codeword corresponds to a separator.

Word-based compression is attractive since it emulates, in a way, higher-order literal compression, i.e., can attain significant compression ratios, while at the same time being simpler to implement and less memory consuming, as the compression model is not polluted with hardly useful contexts. Interestingly, word-based compression even its simplest, static order-0 implementation wins in compression ratio with popular algorithms from Ziv–Lempel family. The property of being static implies a possibility of fast and simple search over a text compressed in that way, and also makes direct access to the text possible (e.g., for decoding and displaying only an excerpt of text from a middle).

Word-based compression is very practical, as long as it can be applied: its mechanism is simple, the search is fast, the compressed text together with its word dictionary takes only about 30% of the original representation [7], and more advanced queries can also be handled with relatively little difficulty. The problem is, however, that the assumption of "text"

* Katedra Informatyki Stosowanej, Politechnika Łódzka

made up of "words" separated with spaces, so natural and convenient for Western languages (e.g., English, French), is inappropriate for oriental languages (e.g., Chinese, Korean), DNA and protein sequences, or structured music files (MIDI). Moreover, word-based approach is not perfect also for some European languages: agglutinative ones, like Finnish or Hungarian, or inflecting ones, like Polish or Russian. For the first group of languages, the vocabulary is potentially infinite, and for the second group, users are often interested in finding any of several grammar forms of a given word (usually having a common prefix). It should be thus clear that there exist important applications for compression algorithms that allow searching directly in the compressed stream, without assuming practically anything about the data. Fredriksson and Grabowski [12] showed how byte codes devised for word-based compression can be adopted to make possible fast search over arbitrary data; this approach will be briefly presented in Sect. 2.

The most natural choice for static order-0 compression on words is Huffman coding [16]. It achieves less than 30% of compression ratio, but the search or decompression are not that very fast because of the need of bitwise manipulations. A more practical idea was used in [17]. They replaced the classic, binary Huffman with 256-ary Huffman. In other words, all Huffman codewords had either 1 bytes, or 2 bytes, etc. This way, decompression got much simpler but searching in the compressed data was still unable to perform any skips over text characters (bytes).

2. Byte codes supporting fast search

The next modification [17] was therefore a *tagged* Huffman code, in which 7 out of 8 bits in each 256-ary Huffman codeword byte carried the actual information about the symbol, and 1 bit was used as a flag to signal the first byte of a codeword. Thanks to it, the tagged Huffman code can be accessed in any position, even if the access point is in the middle of some codeword (examining at most a few following bytes is enough to detect a codeword boundary). Moreover, this scheme enables searching with BM-like algorithms, without any risk of finding false matches, as opposed to plain 256-ary Huffman. The price for all those desirable properties is some deterioration in the compression ratio, to about 35%.

Interestingly, in the family of byte codes with one bit per byte spent for a flag, the tagged Huffman is not optimal. To notice it, it is enough to make a trivial change to the scheme: use a flag to denote the last, not the first, byte of each codeword. In this way, the flag bits in the stream are enough to make the code a prefix one. Consequently, all the combinations on the "message" bits are valid and should be used, which stands in contrast to the tagged Huffman. This idea was presented and analyzed by Brisaboa *et al*. in 2003 [3], under the name of end-tagged dense code (ETDC). Indeed, this code is "denser" than tagged Huffman, but also amazingly simple and easy to implement. The compression ratio improves to 32–33%. Note that what is needed to generate the code is only the list of words ordered by

frequency; the frequencies themselves are not needed. As an interesting historical note, we cite Culpepper and Moffat [8] (2005), on ETDC: *The exact origins of the basic method are unclear, but it has been in use in applications for more than a decade, including both research and commercial text retrieval systems to represent the document identifiers in inverted indexes*.

Traditionally, the flag is the highest bit in a byte (although this is purely conventional) in the described schemes. The solution in ETDC thus means that bytes in the range from 128 to 255 end a codeword, while bytes lower than 128 do not end the codeword. The former values can be called *stoppers* and the latter *continuers*. The ranges for stoppers and continuers is disjoint. What is important, however, is that the value 128 does not have to be the threshold. In general, we can say about having s stoppers and c continuers, with the only condition that $s + c = 256$. This generalization was proposed in [4] under the name of a (s,c)-dense code, or shortly (s,c)-DC. Obviously, ETDC is (128,128)-DC. The authors also use the name (s,c) stop-cont code for a code with each codeword being a sequence of zero or more values from 0 to $c - 1$ terminated with exactly one value from c to $s + c - 1$. In fact, it can be of some value to generalize the (s,c)-DC definition given above with replacing the condition $s + c = 256$ with $s + c = 2^b$, where b is any positive integer (perhaps 4 might be useful for small alphabets, or 16 for huge alphabets), but in the following we assume byte-oriented codes, as most practical, which corresponds to setting $b = 8$.

This (s,c)-DC idea was discovered a year earlier by Rautio *et al.* [19], but their search techniques were different than in [4] and were based on splitting each codeword into two parts sent into separate streams. For typical distributions, s should be greater than c. For example, in the experiments in [7], the optimal s for several large collections of English text varies from 188 to 198, leading to at least 0.5% compression improvement in respect to ETDC. In fact, (s,c)-DC loses very little to plain (non-tagged) 256-ary Huffman, about 0.2% of the size of the original text. Still, the loss to order-0 entropy in the word-based model, which can be achieved by arithmetic coding, is greater, about 4–5%. Unfortunately, it seems impossible to efficiently search in an arithmetically encoded stream. Searching in (s,c)-DC is as easy as in its predecessors, tagged Huffman and ETDC, since (s,c)-DC is also a prefix byte-oriented code, supporting fast Boyer–Moore like searching strategies, increasing their speed with growing pattern length (measured in bytes of the compressed pattern representation).

An interesting question concerns the selection of the s value (while $c = 256 - s$ is then obtained automatically). The code allows for s 1-byte codewords, sc 2-byte codewords, sc^2 3-byte codewords, and so on. Setting a small s implies that the amount of the alphabet symbols (i.e., words, typically) encoded with only 1 byte, will be strongly limited. On the other hand, in such a case the number of symbols represented with 2 bytes (and not 3 or more bytes) will be much larger than for a case of s being significantly larger. Clearly then, the best choice depends on the symbol probability distribution.

According to current knowledge, finding the optimal s value requires a brute-force check for all 127 possible partitions, as there are (rather artificial) distributions for which

the compressed text size as a function of s is not a convex function and thus more than one local minimum exists, making the standard binary search strategy fallible [7]. Fortunately, for real distributions the function is practically always convex.

The search in (s,c)-DC (including ETDC) is slightly different than in tagged Huffman. The latter algorithm uses tag bits to signal the codeword beginnings, i.e., it is impossible to have the pattern misaligned. In this way, false matches are impossible with this algorithm. Now we are going to present on an example what can happen with (s,c)-DC. Let s = 156, i.e. the values [0...9] correspond to continuers, and the values [100...255] represent stoppers. Imagine that the pattern is a single word and is encoded as a pair of bytes (50, 200). Now, let us have a piece of encoded text:

... 35 50 200 ... 187 50 200 ...

Seemingly, there are two matches in the shown piece of text, and both will be found by e.g. BMH algorithm [14], but the first of them should be rejected as being a false one. Indeed, this can be found with peeking just the previous byte. In the first case, the examined value is 35, i.e. belongs to continuers, which means that the found "match" was actually a suffix of some 3-byte or longer codeword. In the second case, the previous byte is 187, i.e. a stopper, which means that found pattern sequence starts at a codeword boundary, i.e. must be a genuine match. This nuisance is very little in fact, since the extra checks are needed only at potential match positions, which are rare.

It seems very hard to improve the compression ratio of (s,c)-DC while keeping its advantages like byte-orientedness and fast search support. Fortunately, it is possible to weaken the input assumptions without losing most of the advantages of the code but improving the compression significantly. The idea, used in the scheme called *pair-based end-tagged dense code* (PETDC) [6], was to treat frequent pairs of adjacent words as individual symbols. PETDC can be classified as a variable-to-variable length code, since the input symbols vary in their length (i.e., the number of component words – one or two) and also vary in the output codeword length. Searching for a pattern in this scheme may bring some problems at the encoded sequence boundaries (since a word can occur alone or be a prefix or a suffix of possibly many word pairs), but this can be solved by referring to a multiple search algorithm. The compression ratio, on standard large English text collections, gets around 28%, in contrast to 32–33% achieved by the original ETDC. The 28% ratio is (accidentally) very similar to order-0 entropy in the word model. The reported compression speed [6] is however about 2.5 times worse than for ETDC, but in the decompression the loss is only about 20%. We are not aware of a similar modification for the (s,c)-DC code, but it is likely that the compression improvement would be very slight.

Another possibility to generate dense codes was pointed out by Culpepper and Moffat [8]. In their solution, the first byte in a codeword keeps information about the length of the whole codeword. Unfortunately, this idea poses trouble with efficient search [9].

Variable-length byte coding on words is also often used for sole text compression. One of the most successful examples of this approach is the *word replacing transform*

(WRT) [21]. The scheme assumes a static dictionary shared by the compressor and the decompressor (or, in other words, by the sender and the receiver), and words not found in the dictionary are written verbatim. The byte-orientedness of the encoding is needed not for searching (which is not supported in WRT, although this option seems possible) but rather for more efficient further compression with general-purpose algorithms. It appears that stronger compressors, e.g., from the PPM family [20], work better with less dense WRT coding, and weaker compressors, e.g. Deflate from the LZ77 family, prefer denser coding. To address this phenomenon, the authors of the cited work devised and implemented two variants of WRT, reaching up to 14% improvement in the latter case, with gzip, when compared to the previous leader, StarNT [24]. Byte encoding on the word level is also one of the main principles in the XML compressor XWRT [22, 23].

Although simple and useful, byte codes are not the only way to go in word-based compression with search support; for alternatives see e.g. [13,15].

Now we will present a way to adopt byte codes (e.g., (s,c)-DC) to work with arbitrary data [12]. To this end, the text is partitioned into non-overlapping q-grams, where the choice of q depends on the text characteristics (e.g., 4 or 5 is a good choice for natural texts in European languages). There are two search algorithms used, one for patterns shorter than $2q–1$ and the other for longer ones. Short patterns are seeked with a modification of Shift-Or automaton [1], which performs implicit decoding of the text, encoded to the automaton [10]. Long patterns must contain at least one full q-gram, and thus the problem boils down to multiple matching over q alignments of the pattern, where each alignment is represented by the concatenation of byte codes of the q-grams it wholly contains. Verifications for the truncated pattern prefix/suffix, and usually also for the multiple matching component, are needed.

3. Denser code for static text

In further considerations we assume that that longest codewords have 3 bytes. This is a reasonable assumption for many data types, including natural language texts. Also, to fix attention, we talk about “words” as symbols of the text, albeit those could be q-grams instead.

Our proposal is based on a very simple observation: some pairs of 1-byte encoded words, although globally frequent, may never go adjacently. For example, in English it is unlikely to come across phrases like “a the”, “the a”, “of of” or “from of”, although all the words in the listed phrases belong to the most frequent ones. Consequently, the pairs of bytes representing those 2-word phrases may never occur in the encoded text. If this is a case, the most frequent words among those encoded on triples of bytes may obtain 2-byte codewords instead, namely those corresponding to the non-occurring pairs of most frequent words.

Let us assume (s,c)-dense coding. There are s^2 pairs of words encoded with one byte each. The text must be scanned and the non-occurring pairs found (this can be implemented

efficiently with e.g. a hash table). Note that, unfortunately, this idea requires the text to be static, i.e., its application is somewhat limited.

Obviously, when codewords longer than 3 bytes occur, this idea remains applicable, but the amount of potential optimizations grows; for example, non-occurring pairs of 1-byte and 2-byte codewords may be used to shift some words encoded on 4 bytes to the group of words encoded of 3 bytes. We have not considered how to solve those issues optimally, however we believe those are mostly of theoretical interest.

The set of non-occurring word pairs must be stored with the encoded text. As it is not very small typically, we found that using s^2 bits to indicate those pairs is more succinct than listing them directly on 2 bytes per pair. For example, with ETDC (s = 128) the resulting overhead is 2 KB.

Finally, we note that this idea makes the coding no longer pure order-0, and as such resembles PETDC a bit. As it will be seen in Sect. 5, our idea gives much humble improvements than PETDC, but is also much easier in implementation and faster in coding and decoding.

Another related idea [11] allows to replace a 2-byte codeword for a word with 1-byte encoding, if only the symbol occurs in an "appropriate" context. Assume that V and W are words having 1-byte encodings according to the baseline code (e.g., (s,c)-DC), and let those codewords be $c(V)$ and $c(W)$, respectively. The phrase $V\,W$ does not occur in the text. Now, alternatively to the previous idea, the 2-byte sequence $c(V)c(W)$ can be used to encode $V\,X$, where X is the most frequent word in the context V among the words encoded on two bytes with the original algorithm. We have not implemented this idea.

4. A denser code with fast synchronization on average

The idea of tags in byte codes (or their more efficient equivalence in (s,c)-DC) is to provide instantaneous synchronization in the compressed stream. On the other hand, it is known that e.g. in Huffman sequences synchronization delays after a random access may be arbitrarily long (although with probability of such an event going to zero with the number of decoded bits going to infinity), but also it is known that in most cases the delay is rather short. There are ways to limit the synchronization delay in the worst case for Huffman codes, for the price of adding very small redundancy in practice; see e.g. [2]. We are not aware about similar results for byte codes.

Now we propose a very simple modification to the (s,c)-dense code, which improves the compression ratio for the lack of guarantee of $O(1)$-time synchronization. Still, as we will see, the synchronization delay is $O(1)$ on average.

Let us consider a probability distribution for which the longest (s,c)-DC codeword has k bytes, and $k \geq 3$. The group of longest codewords has then at most c^{k-1} elements. The proposed modification increases its capacity to $c^{k-1} \cdot 256$, without reducing the other groups. Note that after this modification the optimal s may be somewhat greater, thus allowing some symbols obtain shorter codewords.

To fix attention, assume $k = 3$. The encoded text consists of 1-byte, 2-byte and 3-byte encoded symbols, where the codewords in the respective groups are $s^{(i)}$, $c_1^{(i)}s^{(i)}$, and $c_1^{(i)}c_2^{(i)}b^{(i)}$, where $s^{(i)}$ belong to [c...255], $c_1^{(i)}$ and $c_2^{(i)}$ to [0...c – 1], and $b^{(i)}$ belong to [0...255]. Assume now that we have found a tentative match somewhere in the encoded text, and the byte just preceding the match location is from [0...c – 1] range. With the standard (s,c)-DC, we would know this is a false match, but now it can also be the last byte of a 3-byte encoded word (i.e., a genuine match). We decode two more bytes back; if any of them is not from [0...c – 1], then we know that our match is a false one and the verification terminates. If both bytes are however from [0...c – 1], then we have to continue peeking previous bytes. The procedure ends (with positive or negative result) upon encountering the first byte from the stopper range, i.e., [c...255]. It can be shown easily, assuming Zipfian distribution of words in the text, that the probability that a randomly extracted word from the text is encoded on 3 bytes, is upper-bounded by $1 - \varepsilon$, where $\varepsilon > 0$ (in practice, Í is about 2/3 or more). Assuming also lack for correlation between the codeword lengths of successive words, we obtain that the probability of a k-long run of 3-byte encoded words in a randomly pointed location of the encoded text is upper-bounded by $(1 - \varepsilon)^k$, which tends to zero with growing k, and the expected value of k is $O(1)$.

5. Experimental results

We have implemented the first idea from Sect. 3 and tested its effectiveness in the word based and q-gram based models. For the word model we used the following files, used earlier in [12], and available in the archive http://szgrabowski.kis.p.lodz.pl/research/data.zip: Dickens (the collected works of Charles Dickens, 10192446 bytes); XML (collection of XML files, 5345280 bytes); English and Spanish versions of the Bible (4486219 and 4276390 bytes, respectively). In the XML file, the end-of-line symbols were inconsistent, so before further processing we converted 2-byte EOLs to #10. Additionally, we took a couple of larger files from the Pizza & Chili Corpus [18]: those are 50 MB excerpts (prefixes) of English texts, source codes, and XML, and are referred to in tables as *English50*, *Sources50* and *XML50*. For the second experiment, with q-gram models, we additionally took 50 MB datasets of MIDI pitch values and proteins from Pizza & Chili, which are denoted here as *Pitches50* and *Proteins50*. The compression results are presented in Table 1 and Table 2, respectively.

For compression on words, the spaceless model was used. The vocabulary itself was *zlib*-compressed (with the maximum setting, -9), following [12]. Clearly, better dictionary encoding schemes are possible but this requires a separate study. The words, given as input for *zlib* compression, were ordered according to their rank and separated with #1 symbols, which did not occur in any of the given files. Identical vocabulary compression was used for q-grams but this time the separators were not used for obvious reasons.

The improvements on words are rather disappointing, especially for ETDC. In one case (the English Bible) the modification resulted even in a tiny loss (about 20 bytes). This happened because the sub-vocabulary of 3-byte encoded words had only around 2,000 entries for this file, and thus not the whole gained code space could be utilized.

Table 1
Comparison of compression ratios in word based schemes

	gzip -9	bzip2 -9	W-ETDC	denser W-ETDC	W-(s,c)-DC	denser W-(s,c)-DC
Dickens	37.79%	27.47%	34.80%	34.65%	33.79% (s=206)	33.24% (s=206)
Bible, English	29.54%	21.15%	32.43%	32.44%	31.12% (s=219)	30.92% (s=219)
Bible, Spanish	31.59%	23.28%	39.43%	39.19%	38.33% (s=205)	37.76% (s=205)
XML (5 MB)	12.39%	8.25%	37.30%	37.11%	35.52% (s=228)	35.03% (s=228)
English50	37.52%	28.40%	35.99%	35.74%	35.26% (s=197)	34.53% (s=197)
Sources50	23.29%	19.78%	46.24%	45.78%	45.22% (s=201)	44.05% (s=201)
XML50	17.23%	11.17%	35.66%	35.35%	35.19% (s=202)	34.47% (s=202)

With (s,c)-DC, the gains are greater which is due to two reasons. One is that the parameter s is greater than 128, so among the s^2 word pairs there are more such that never occur adjacently in the text. The second reason is that with a greater s, there are more items in the group of 3-byte encoded words, hence the most frequent of them have more occurrences than it used to be with ETDC.

The q value most often used in the experiment was 4. In [12] it was found the best compromise for natural language texts; q=5 may provide still a bit better compression (in spite of the bloated dictionary), but the minimum pattern length for which the faster of the two possible algorithms is selected, is then greater. For some data types other values of q are more suitable though. For very redundant XML data we present results for q = 4 and q = 10. Proteins and MIDI pitches are much less compressible, hence q = 3 was also tested for them. In both tables, gzip and bzip2 compression ratios are also given, for a reference.

The implementations were written in Python and tested on an Athlon64 3000+ with 2 GB of main memory. Implementation of the idea from Sect. 4 will require pattern search speed measurements too, for a honest evaluation, so future works must require switching the programming language to e.g. C++.

Table 2
Comparison of compression ratios in q-gram based schemes

	gzip -9	bzip2 -9	q-ETDC	denser q-ETDC	q-(s,c)-DC	denser q-(s,c)-DC
Dickens, q=4	37.79%	27.47%	49.01%	47.90%	48.69% (s=166)	47.08% (s=166)
Bible, English, q=4	29.54%	21.15%	47.03%	46.33%	46.47% (s=179)	45.33% (s=179)
Bible, Spanish, q=4	31.59%	23.28%	48.73%	47.91%	48.21% (s=177)	46.88% (s=177)
XML (5 MB), q=4	12.39%	8.25%	48.59%	47.81%	47.50% (s=199)	45.83% (s=199)
XML (5 MB), q=10	12.39%	8.25%	35.91%	35.27%	35.81% (s=166)	34.90% (s=166)
Proteins (5 MB), q=3	56.63%	53.27%	64.02%	64.06%	62.50% (s=221)	62.61% (s=221)
Proteins (5 MB), q=4	56.63%	53.27%	70.33%	65.62%	70.33% (s=131)	65.43% (s=131)
English50, q=4	37.52%	28.40%	49.54%	48.17%	49.25% (s=162)	47.30% (s=162)
Sources50, q=4	23.29%	19.78%	56.03%	54.03%	55.87% (s=154)	53.21% (s=154)
XML50, q=4	17.23%	11.17%	43.02%	41.92%	42.17% (s=182)	40.28% (s=182)
XML50, q=10	17.23%	11.17%	34.40%	34.01%	34.17% (s=180)	33.50% (s=180)
Pitches50, q=3	30.59%	36.12%	71.48%	69.06%	71.34% (s=147)	68.28% (s=147)
Pitches50, q=4	30.59%	36.12%	71.92%	70.11%	71.89% (s=140)	69.77% (s=140)
Proteins50, q=3	47.39%	45.56%	63.99%	64.00%	62.73% (s=217)	62.69% (s=217)
Proteins50, q=4	47.39%	45.56%	65.40%	61.02%	65.39% (s=133)	60.72% (s=133)

6. Conclusions

We presented a few ideas improving the compression ratio in known dense byte codes. Experiments with one of those ideas involved ETDC and (s,c)-DC codes, but tagged byte Huffman could also be used. The achieved improvements are greater in case of q-gram based, rather than word based, compression. Still, in no case the gains are spectacular, but

we defend our (implemented) idea as being very simple both conceptually and programmatically, and not affecting search or decompression speed. How practical are the remaining ideas presented in the paper remains task for future work.

Acknowledgement

The author wishes to thank Kimmo Fredriksson for pointing out the idea of succinct word encoding in some contexts, presented in Sect. 3.

References

[1] Baeza-Yates R.A., Gonnet G.H., *A new approach to text searching*. Commun. ACM, 35(10), 1992, 74–82.

[2] Biskup M.T., *Guaranteed Synchronization of Huffman Codes*. Proc. Data Compression Conference (DCC'08), 2008. IEEE Computer Society Press, 462–471.

[3] Brisaboa N., Iglesias E., Navarro G., Paramá J., *An Efficient Compression Code for Text Databases*. Proc. ECIR'03, LNCS 2633, 2003, 468–481.

[4] Brisaboa N., Fariña A., Navarro G., Esteller M., *(s,c)-dense coding: An optimized compression code for natural language text databases*. [w:] Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003), LNCS 2857, Springer-Verlag, 2003, 122–136.

[5] Brisaboa N., Fariña A., Navarro G., Paramá J., *Simple, fast, and efficient natural language adaptive compression*. [w:] Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE 2004), LNCS 3246, Springer-Verlag, 2004, 230–241.

[6] Brisaboa N., Fariña A., Navarro G., Paramá J., *Improving semistatic compression via pair-based coding*. [w:] Proc. 6th International Conference on Perspectives of System Informatics (PSI'06), LNCS 4378, 2006, 124–134.

[7] Brisaboa N., Fariña A., Navarro G., Paramá J., *Lightweight natural language text compression*. Information Retrieval, 10, 2007, 1–33.

[8] Culpepper J.S., Moffat A., *Enhanced byte codes with restricted prefix properties*. [w:] Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE 2005), LCCS 3772, Springer-Verlag, 2005, 1–12.

[9] Culpepper J.S., Moffat A., *Phrase-based pattern matching in compressed text.* [w:] Proc. 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006), Springer--Verlag, 2006, 337–345.

[10] Fredriksson K., *Shift-or string matching with super-alphabets*. Information Processing Letters, 7(4), 2003, 201–204.

[11] Fredriksson K., Private communication, Feb. 2007

[12] Fredriksson K., Grabowski S., *A general compression algorithm that supports fast searching*. Information Processing Letters, 100(6), 2006, 226–232.

[13] Fredriksson K., Nikitin F., *Simple compression code supporting random access and fast string matching*. [w:] Proc. 6th Workshop on Efficient and Experimental Algorithms (WEA'07), LNCS 4525, Springer-Verlag, 2007, 203–216.

[14] Horspool R.N., *Practical fast searching in strings*. Software–Practice and Experience, 10(6), 1980, 501–506.

[15] Klein S.T., Kopel Ben-Nissan M., *Using Fibonacci compression codes as alternatives to dense coding*. [w:] Proc. Data Compression Conference (DCC'08), IEEE Computer Society Press, 2008, 472–481.

[16] Moffat A., *Word-based text compression*. Software–Practice and Experience, 19(2), 1989, 185–198.
[17] de Moura E., Navarro G., Ziviani N., Baeza-Yates R., *Fast and flexible word searching on compressed text*. ACM Transactions on Information Systems (TOIS), 18(2), 2000, 113–139.
[18] Pizza & Chili Corpus. Maintained by P. Ferragina and G. Navarro. 2005–2007. http://pizzachili.dcc.uchile.cl/index.html.
[19] Rautio J., Tanninen J., Tarhio J., *String matching with stopper encoding and code splitting*. [w:] Proc. of Combinatorial Pattern Matching (CPM), LNCS 2373, Springer-Verlag, 2002, 42–52.
[20] Shkarin D., *PPM: One step to practicality*. [w:] Proc. Data Compression Conference (DCC'02), IEEE Computer Society Press, 2002, 202–211.
[21] Skibiński P, Grabowski S., Deorowicz S., *Revisiting dictionary-based compression*. Software–Practice and Experience, 35(15), 2005, 1455–1476.
[22] Skibiński P., Grabowski S., Swacha J., *Effective asymmetric XML compression*. Software–Practice and Experience, 2008, accepted.
[23] Skibiński P., Swacha J., Grabowski S., *A Highly Efficient XML Compression Scheme for the Web*. [w:] Proc. 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'08), LNCS 4910, 2008, 766–777.
[24] Sun W., Zhang N., Mukherjee A., *A Dictionary-Based Multi-Corpora Text Compression System*. [w:] Proc. of Data Compression Conference (DCC'03), IEEE Computer Society Press, 2003, 448.